UiO : Department of Informatics
University of Oslo

Automated cloud bursting on a hybrid cloud platform

Evaluating and prototyping automated cloud bursting and hybrid cloud setups using Apache Mesos

Noha Xue

Master's Thesis Spring 2015

Automated cloud bursting on a hybrid cloud platform

Noha Xue

May 18, 2015

Abstract

Hybrid cloud technology is becoming increasingly popular as it merges private and public cloud to bring the best of two worlds together. However, due to the heterogeneous cloud installation, facilitating a hybrid cloud setup is not simple.

In this thesis, Apache Mesos is used to abstract resources in an attempt to build a hybrid cloud on multiple cloud platforms, private and public. Viable setups for increasing the availability of the hybrid cloud are evaluated, as well as the feasibility and suitability of data segmentation.

Additionally an automated cloud bursting solution is outlined and implementation has been done in an attempt to dynamically scale the hybrid cloud solution to temporarily expand the resource pool available in the hybrid cloud platform using spot price instances to maximize economical efficiency.

The thesis presents functional and viable solutions with respect to availability, segmentation and automated cloud bursting for a hybrid cloud platform. However, further work remains to be done to further improve and confirm the outlined solution, in particular a performance analysis of the proposed solutions.

Acknowledgements

I would like to offer my special thanks to my supervisor, Hårek Haugerud, for guidance and encouragement during the thesis work. His opinions and constructive suggestions given during our discussions have been of great help and are greatly appreciated. Without his support, the thesis would not have been as improved to its current shape and form.

My special thanks to Lars Haugan for introducing me to interesting emerging technologies, including Apache Mesos which ended up being a central piece of technology in my thesis.

I also wish to acknowledge the guidance provided by Kyrre Begnum, in particular for his lectures and inspiring assignments during the master program which prepared me for my thesis work. His work with managing and keeping Altocloud in tip top shape is also greatly appreciated.

To my friends, fellow students, and in particular my family, I would like to express my endless gratitude for the support both in academia and in general.

Finally, I wish to express my sincere appreciation to everyone, who directly or indirectly, have lent their support to me in any manner or form.

Thank you,

Noha Xue

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The use of cloud computing is becoming more common, bringing along the advantages of flexibility and abundance of available resources, but also a higher degree of complexity along with privacy and security concerns.

Nevertheless, the cloud technologies progresses and matures each year, providing functionality for individuals for personal use, as well as enterprises with huge requirements to performance, availability, and price.

According to the report, *State of the Market, Enterprise Cloud 2014* published by Verizon Enterprise Solutions (2014), 65% of enterprises are using cloud services, with 71% of them expecting to use the cloud for external-facing applications by 2017.

However, some data storage and processing may be more sensitive and require restrictions to on-site data centers or approved cloud providers due to laws or confidentiality policy. An example of this would be processing of highly sensitive information, that would be required by law to only take place within the residing country of the company.

In 2013, Edward Snowden disclosed a vast amount confidential documents regarding USA's governmental institution National Security Agency's surveillance operations and capabilities, which raises concerns about the risks of storing sensitive information at external cloud locations.

Additionally, the use of a single cloud provider may be problematic in terms of vendor lock-in as well as price and availability. Even the largest cloud companies experience downtime which may have an adverse effect on the service, depending on the requirements and system design.

In 2014, Amazon Web Services, one of the biggest and most established cloud provider, experienced 23 outages on *Elastic Compute Cloud* (EC2) resulting in 2.69 hours of downtime (Shado, 2015). While the *Service Level Agreement* (SLA) ended up being an impressive 99.9974%, there was still other parts of Amazon's services that was affected.

Amazon's *Content Delivery Network* (CDN) service CloudFront experienced issues that resulted in downtime for approximately 90 minutes in November 2014 (Silasi, 2014).

Furthermore, SLA alone does not by default guarantee anything but the availability of the service and by extension does not guarantee any set *Quality of Service* (QoS), which may adversely affect the performance of an application to the unacceptable levels.

A possible solution for this is to utilize multiple cloud providers to minimize the risks of service disruption and degradation. There are several organizations, like MODAClouds, that are working with solutions for providing interoperability between different cloud providers (MODAClouds, n.d.).

Additionally, a platform utilizing multiple clouds will also allow organizations to be able to pick one or several cloud providers based on various of factors like price, location, availability, and performance among many others.

An another viable option would be the use of private data centers in addition to external cloud providers in a hybrid setup. This type of setup is often referred to as *hybrid cloud* and is becoming increasingly popular as more companies are starting to invest and offer these solutions. According to *Google Trend Search*, there has been an increasing interest in the term, *hybrid cloud*, since the beginning of 2009 (Google Inc., 2015).

However, most solutions on the market are either proprietary or not open-sourced which is not ideal if the purpose of using multiple cloud providers is to avoid vendor lock-in.

Even with the possibilities of using multiple cloud providers and private data centers, there is still the issue of static partitioning and isolation of resources due to the design of virtualization. Static partitioning of systems makes it difficult to fully utilize the resources due to fluctuations in system use which may be affected by various factors like business hours, holidays, and batch processing just to name a few.

Clustering technology as an abstraction layer on top of cloud resources is one way to solve the issue of static partitioning, with the possibility to turn scattered hardware into a flexible platform to be able to dynamically allocate resources depending on needs at the time, thus facilitate a more efficient use of the resources.

Most frameworks working on top of clustering technologies seem to allow for restrictions on how and where an application can be executed based on set attributes. This could solve the issues regarding the concerns of the confidentiality and the wish to segmented sensitive data and processing based on some specified restrictions.

Additionally, Amazon Web Services offers an interesting payment model for instances called *spot instances* (Amazon Web Services, Inc., n.d.-a). These instances uses the excess capacity Amazon Web Services have available and the prices for spot instances fluctuates according to supply-and-demand hourly. There are some considerations that needs to be addressed when using spot instances, with the possibility of sudden termination being the main concern. The use of spot instances requires fault-tolerance mechanisms in order to be beneficial for running workloads with SLA or QoS requirements.

This thesis will explore and document the attempts at designing and and prototyping one or several possible solution for constructing a computer cluster built on top of a private servers and external cloud providers. Additionally, investigations will be conducted on the possibilities utilizing spot instances for cloud bursting purposes and for segmenting applications based on certain parameters as well as high availability solutions, leveraging multiple clouds as a possible way to further minimize downtime.

1.1 Problem statement

The following problem statement will used as the foundation for this thesis.

How can we build highly available, segmented computer clusters using private computer hardware together with public cloud providers as a hybrid cloud platform, leveraging spot price instances for an automated cloud bursting solution?

For the purpose of the thesis, several assumptions and definitions are made to narrow down the scope of the project. Here follows a short breakdown of some terms used in the problem statement. Additional clarifications will be found in the *Background chapter 2* and *Approach chapter 3*.

- *Highly available* alludes to high levels of service uptime and availability. The solution should be available for a legitimate user with as little service disruption as possible. The solution should, by combining the characteristics of a computer cluster and multiple cloud providers further lower the risk of service downtime beyond the levels of a single cloud provider.
- *Segmented* denotes the possibility to divide the cluster into logical segments based on certain variables like location, performance, and any other desired factor. The segments should be isolated from each other. Furthermore it also refers to the ability for the solution to restrict tasks to be run on those divided segments.
- *Spot price instances* are instances available for a changing price levels that may be terminated immediately should the current price exceed the offered price for the instances.
- *Cloud bursting* refers to the use the cloud to handle spikes in processing needs. An organization can with cloud bursting dimension their

data center for average workloads and only pay for additional processing during spikes using public cloud providers.

Chapter 2

Background

2.1 Clustering

Computer *clustering* can loosely be defined as a group of hardware connected together to provide a single virtual and powerful platform of hardware. Clustering setups allows developers to leverage a multitude of types of machines as a single platform. This abstracts the hardware layer, rendering the hardware as not essential by itself and can easily be swapped out either due to performance needs or condition.

Having a single and uniform interface towards a cluster makes it possible for flexible partitioning of the resources. A typical workload of a front facing web server would be high traffic during working hours and little during night time. In the case of static partitioning, a set amount of resources is dedicated to that particular task and it is difficult to efficiently use the idle resources during the night time for other purposes. Additionally, in order to deal with peak level of traffic during working hours, the resources granted would be overprovisioned for the rest of the time.

With the added layer of abstraction clustering gives, it is possible to partition the hardware dynamically using software. Services running on top of a cluster can therefore dynamically scale and move within the cluster without being limited by the underlying hardware partitioning. Large companies leverages this type of flexible partitioning for their services. Google with their self-developed platform *Omega* and Twitter with Apache Mesos (Wilkes, 2014; Schwarzkopf, Konwinski, Abd-El-Malek, & Wilkes, 2013; The Apache Software Foundation, 2015b).

2.1.1 Apache Mesos

Apache Mesos is a distributed system kernel that abstract hardware resources like CPU, memory and storage to construct a dynamically

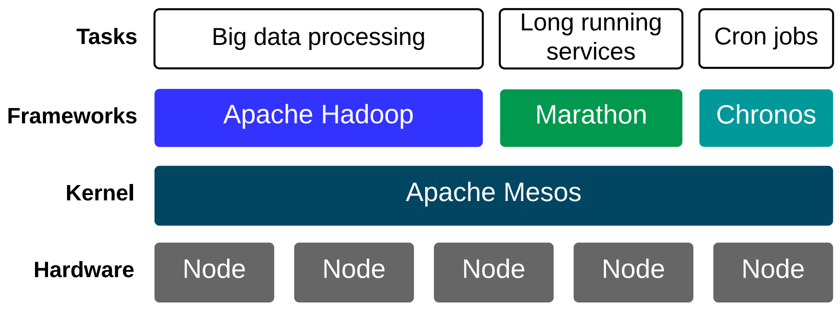


Figure 2.1: Abstraction model of Apache Mesos and some related frameworks.

partitioned computer cluster. Due to similarities in how an operating system abstracts hardware, Apache Mesos has been referred to as a "datacenter OS" (Leopold, 2015).

The Apache Mesos project was initially started as a research project at the University of California, Berkeley by eight students, including one of the founders of Mesosphere, Benjamin Hindman (Zaharia et al., 2011; Mesosphere, Inc., n.d.-b). The goal of the project was to create a system to allow for fine-grained resource sharing in computer clusters (Hindman et al., 2011).

Apache Mesos provides a uniform computer environment for operators and developers to work against without the need to consider the underlying hardware setup. This works by adding an abstraction between the hardware and software frameworks and handling resource allocation between those parts as seen in Figure 2.1.

There are various parts of Apache Mesos working to provide a functioning and robust master-slave setup, as well as scheduling and executing tasks given by a framework in the distributed environment.

Coordination of master node and fail-over

Apache ZooKeeper is the subsystem responsible for coordinating the election of master nodes and to manage fail-over should a master node fail to respond. This subsystem is what provides redundancy for the master nodes in Apache Mesos.

In essence, Apache ZooKeeper provides the tools and means to coordinate distributed applications, providing distributed synchronization, leader election, and group services to name a few.

The slave nodes by themselves are dispensable and the cluster will function as long as there is a adequate level of resources available.

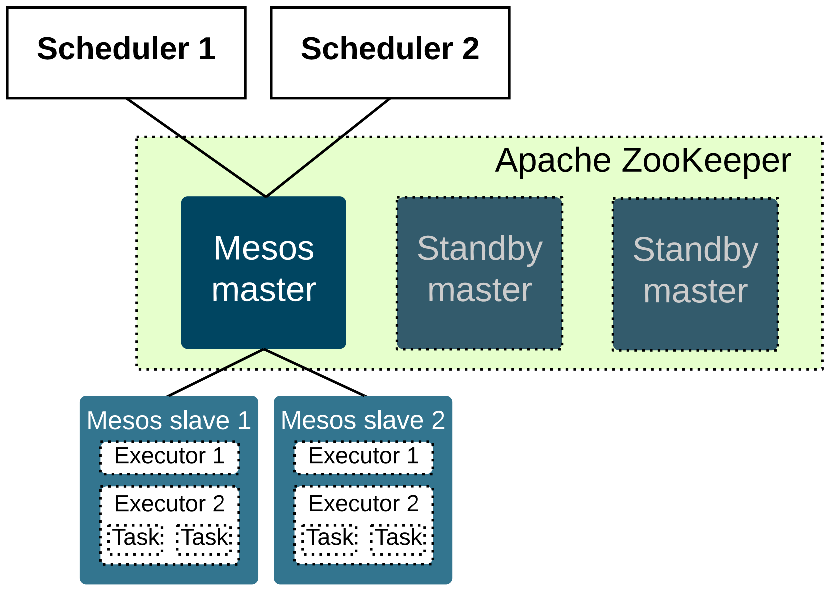


Figure 2.2: A simplified model of how tasks are scheduled and executed.

Resource management

Apache Mesos operates with a layered resource negotiation handled by an allocation module located at the master node. A framework specific scheduler will receive and process resource offerings, while a framework specific executor running at the slaves will allocate resources and launch tasks given. The general setup can be viewed in Figure 2.2.

An example of steps conducted for negotiating the resources and tasks can be summarized briefly as following:

1. A slave node will generate a report of the available resources on the machine and send it to the Mesos master.
2. The allocation module at the master node will, according to some predefined priority, send a resource offering to a framework through the scheduler.
3. The framework will then either accept or reject the resource offer.
    - If the framework accept some or all the resources, it will generate the tasks and send it to the Mesos master.
    - Otherwise, the Mesos master will send the resource offer to the next framework.
4. The Mesos master forwards the tasks to the particular slave node which offered the resources.
5. The slave node receives the tasks and the executor allocates the necessary resources and launches the task.

Encapsulation

Apache Mesos can be deployed directly on a Linux distribution, like for instance CentOS or Ubuntu. Apache Mesos supports most major Linux distributions and provides isolation for the running tasks using Linux containerization utilizing a Linux kernel feature, *Control groups* (cgroups).

cgroups limits and isolates resources like CPU, memory, disk I/O, network on processes. Apache Mesos uses it to encapsulate the executors from each other, thus preventing different frameworks from colliding and interfering with each other.

Additionally, Apache Mesos also supports the use of Docker as the encapsulation mechanism.

Alternatives

Yet Another Resource Negotiator (YARN), was developed as a resource manager and scheduler for the next generation of Hadoop. YARN can be looked upon as a competitor of Apache Mesos, as both technologies attempts to solve the same problem, albeit with different strategies. Apache Mesos utilizes a two-level resource scheduling strategy and was developed as general type of scheduler, while YARN opts for a monolithic approach mainly focusing on Hadoop. However, despite the technologies being similar and arguably competing, an Apache Mesos framework that utilizes YARN has been developed called *Myriad* (Mesos, n.d.).

Usage

Apache Mesos is being used by several large and well known companies for various purposes. Airbnb, eBay, Groupon, Netflix, and Uninett are just a few, with Twitter as one of the main driving forces behind the development of Apache Mesos (The Apache Software Foundation, 2015c; Twitter, Inc., 2013). Twitter embraced Apache Mesos after a conference talk by Benjamin Hindman in March 2010, with a few of Twitter's engineers having previously worked at Google. They missed the capabilities of Google's own clustering solution, Borg, the predecessor of Omega, Google's new clustering solution, and saw an opportunity to shape Apache Mesos into an alternative solution for this (Metz, 2015). Today Twitter uses Apache Mesos for various of their core services in production. Interestingly, Benjamin Hindman restarted a Mesos master running critical production Twitter services during a demo at AMPLab at University of California, Berkeley in 2012, having enough confidence in the Apache Mesos to risk considerable consequences (UC Berkeley AMPLab, 2012).

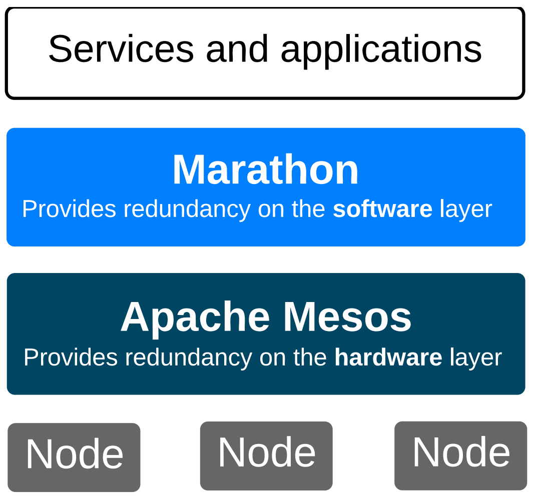

Services and applications
Marathon
Provides redundancy on the **software** layer
Apache Mesos
Provides redundancy on the **hardware** layer
Node
Node
Node

Figure 2.3: The layered components that provides redundancy.

Currently, Apache Mesos is very early in its developing stages with current version being 0.22.1 *Transport Layer Security* (TLS) support is under development and is currently staged to be released in version 0.23.0 (The Apache Software Foundation, 2015a). The rapid development in its initial stages shows that this is a project still in its inception and yet, is being used in production environments in large companies like Twitter.

Mesos frameworks

The frameworks are what provides utility to the cluster, also referred to as *Mesos applications*. Through the schedulers the frameworks receive resource offers and submit tasks to be launched.

Marathon on top of Apache Mesos provides a robust platform for running long time services, making it easier to achieve high SLA and to scale services. With Apache Mesos managing hardware redundancy, Marathon does the same for applications. The Marathon framework will ensure that the specified applications are running as long as Marathon is running, with Apache Mesos ensuring that Marathon is running as long as there is a bare minimum of nodes running and idle resources are available. See Figure 2.3 for an illustration.

Hadoop, a popular distributed processing framework, can be run on top of Apache Mesos. As opposed to a standalone Hadoop cluster, Apache Mesos provides an easier and a more flexible way of managing the cluster itself. Additionally, the cluster can be shared with other frameworks for an even more efficient use of the resources.

Chronos was created as a distributed version of *Cron* and just like Cron, schedules and runs jobs at specified intervals. This framework utilizes the

Mesos cluster to provide a redundant and fault-tolerant service to execute batch jobs.

There are many more frameworks available for Apache Mesos and it is possible to create your own framework for Apache Mesos using the primitives provided.

2.2 Cloud computing

Cloud computing has over the past few years taken off in popularity and availability. Instead of handling acquisition of hardware along with the cost, installation, and maintenance needed to keep an IT infrastructure running, it is now possible to rent virtual machines with a cloud provider and only pay for the resources used, thus making it possible to avoid capital expenditure. Companies are looking towards the cloud for cutting costs and improving their services.

With the ability to manage virtual machines at the cloud or *instances*, which they are commonly referred as, one can within a short amount of time add or remove instances. This flexibility brings along numerous possibilities. By clustering together the instances, one could scale a service according to various variables like demand, cost, power efficiency or any other one see fit.

Cloud services are mainly provided according to one of three service models:

- *Infrastructure as a Service* (IaaS)

  Provides resources as physical or virtual hardware. Mature cloud providers using this service model often has a rich feature set, which gives the possibility to control and change storage devices, the underlying network, and other underlying mechanisms one need to be able to control in order to emulate a data center.

  The users of these type of cloud services needs to install and maintain everything that is needed for their purposes on top of the virtual hardware. This would include setting up and maintaining operating systems, software, and security among others.

- *Platform as a Service* (PaaS)

  In this service model, an environment is provided to the user for running applications and services. This commonly includes web services, databases and runtime environments for executing software.

  Resources in this service model is often allocated dynamically and scales according to the required levels, without the need for users to manually maintain the resources.

- *Software as a Service* (SaaS)

  SaaS refers to the service model that provides software and services to the users directly. The users interact with software directly and do not need to install and maintain the hardware and service needed to maintain the software. Typical software available through this service model would be Office 365, Gmail, and the CRM solutions provided by Salesforce just to name a few.

2.2.1 Altocloud

Altocloud is the name of the OpenStack installation located at Oslo and Akershus University College of Applied Sciences. The installation is running the Havanna release, two versions prior to the current release, Juno.

For the purposes of this thesis, Altocloud will function as a role of a private data center. The OpenStack provides an interface to manage virtual machines, which in this thesis, will be used to build part of the cluster.

2.2.2 Amazon Web Services

Amazon Web Service (AWS) is the name of the collection of cloud services provided by Amazon Web Services, Inc. with *Amazon EC2* and *Amazon S3* in the center as the key services among many others. AWS is one of the first cloud providers that emerged, is one of the most mature cloud provider, with a reach feature set, and has an impressive amount of known companies using their services.

The abbreviation for Amazon S3 is Amazon Simple Storage Service and provides cheap storage for use in combination with other Amazon services or standalone. Amazon S3 is primarily used for bulk storage and when more persistent type of storage is desired.

Amazon EC2 is an abbreviation of Amazon Elastic Computing Cloud and provides computer resources in what essentially are virtual machines. Many different levels of instances are offered each with their own resource priorities, such as compute optimized, *M3* and memory optimized, *R3*.

A very interesting pricing scheme which if offered for Amazon EC2 is the *spot instances* (Amazon Web Services, Inc., n.d.-a). Unused EC2 capacity is put out for bidding and the price will fluctuate based on the demand. It works by creating spot instances and only paying for the current price up until a self-specified maximum bid. If the current price should exceed the maximum bid, the spot instances will be terminated with a 2 minute warning. For setups with proper fail-over procedures this type of instances may be very attractive for certain workloads.

Even with proper a fail-over solution, there is a cost associated with fault-handling a task. An unfinished task running for a set amount of time is wasted use of resources. Voorsluys and Buyya (2012) highlights this and looks into the pricing system and proposes a solution for determining the optimal price with respect to cost of fault-handling the tasks that was lost in instance termination. The paper outlines 5 different bidding strategies which may be employed.

Furthermore, according to Agmon Ben-Yehuda, Ben-Yehuda, Schuster, and Tsafrir (2013), the prices are not purely based on the supply-and-demand, but also involves an additional specific value calculated from an algorithm at Amazon Web Services.

2.2.3 Terms and implementation models

Hybrid cloud

The term *hybrid cloud* is fairly well defined and denotes the use two or more distinct cloud platforms, usually an in-house and private cloud platform in combination with a public and third-party cloud platform to perform some set workload (Mell & Grance, 2011; Interoute Communications Limited, n.d.; Apprenda Inc., n.d.; Sanders, 2014; Bittman, 2012). Private clouds provide security, privacy, and control of data stored and processed there, low latency due to locality, and prevents service disruption due to external networking problems. The use of public cloud services are cheap, does not require capital expenditure, are easily scalable, and applications can be deployed on different geographic locations for extra redundancy. Hybrid cloud platforms combines these advantages and attempts to increase the degree flexibility as well as control of the data. Additionally, IP address ownership is one of the issues with using public cloud providers that can be mitigated by using hybrid clouds. Moving away from a public cloud provider may pose problems if the service is highly coupled to the IP addresses.

Cloud bursting

A specific workload deployment model called *cloud bursting* utilizes a hybrid cloud solution to load balance a workload between private computer resources and public clouds (Mell & Grance, 2011; Nair et al., 2010). In this model, workloads are mainly processed in-house using private resources, with the possibility to "burst" out into public cloud providers should the workload be too much for the in-house resources to handle. This allows an organization to dimension their data center for average workloads and deal with the spikes using the public cloud and only pay for the extra computer resources when used.

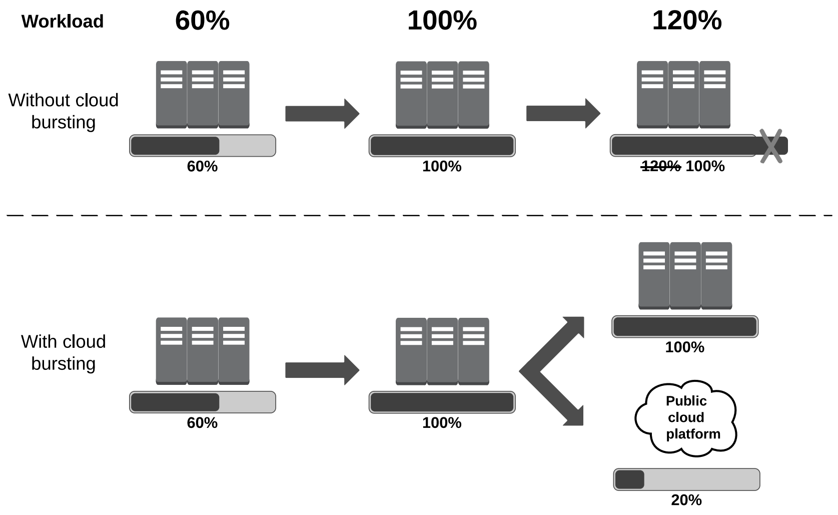


Figure 2.4: Illustration of how cloud bursting works when the capacity limit of the private data center is exceeded.

Spot prices instances

Spot price instances are for Amazon Web Services EC2, instances that are running on surplus resources of the availability region. Currently, there are no other public cloud providers that are offering the same price scheme.

Spot price instances are leased out at a very cheap price, often amounting to 40-50 percent off the on-demand price of the same instance type. The spot prices fluctuates depending on the current supply and demand, with the top bidders getting the spot instances. The prices is set to the lowest winning bid and even if a maximum bid of 1.000 USD has been made, if the lowest winning bid is 0.010, only 0.010 will be billed.

Due to an interesting mechanic of the spot instances offered at Amazon Web Services, should EC2 be the cause of instance termination, no charge will be billed for any interrupted hour (Amazon Web Services, Inc., n.d.-a). This means that if a spot instance is terminated after 50 minutes since start, the cost for that partial hour will be waived, ultimately resulting no charge for that spot instance. Termination initiated by the user will be billed for every partial hour used, just as normal on-demand instances.

Additionally, when a spot instance request has been made, it can not be modified, only canceled. Consequently, the only way to change a price offer is to terminate the spot requests, along with the actual spot instances and re-request them at the new price level desired. Furthermore, once a spot price instance has been marked for termination, it can not be prevented. This is most likely to a countermeasure to prevent complete auto-adapting solutions, which would affect the the spot price market considerably. An example of such a scenario is the *trader bots* in the the stock markets where undesired noise and large spikes occurs due to the high-speed trading.

2.3 Related work

The concept of *multicloud* and *hybrid cloud* is not new and several companies are venturing out to explore and capitalize these concepts.

Multicloud is defined as something that utilizes multiple cloud platforms to run a set of tasks. The definition is still somewhat fuzzy partly because of how new the concepts are and how little the word has been used in general. For the purpose of this thesis, *multicloud* will be defined as the use of multiple externally provided cloud platforms in tandem which have separate *Service Level Agreements* (SLA). For instance, using AWS in combination with a private OpenStack installation.

A project that is looking into multicloud scenarios specifically is *MODA-Clouds* (MODAClouds, n.d.). The project lists up various reason for why the use of multiple clouds in advantageous and is working on a several tools to provide an environment for utilizing multiple cloud providers. By the end of 2015, MODAClouds aims to provide methods and tools for developing multicloud applications, run them and provide quality assurance. The project has collaborators from many countries in the Europe, including the Imperial College of Science in London, Sintef in Norway, and Siemens to name a few.

Several large companies are offering hybrid cloud solutions, aiming to provide a seamless experience, and often in conjunction with existing product portfolio. VMWare is offering a hybrid cloud solution called *vRealize suite* which provides one interface to manage the entire hybrid cloud platform (VMWare, Inc., n.d.-b, n.d.-a). vRealize supports numerous of public cloud providers and private cloud solutions and makes it easy to manage it through their interface. This solution is proprietary and may pose some problems in terms of vendor lock-in. There are other companies that are looking into or are already offering hybrid cloud solutions where they have geographic presence, like *Cisco*, *IBM* and *RackSpace* just to name a few well known companies (Butler, 2015; IBM, n.d.; Rackspace, Inc., n.d.). In a paper written by Breiter and Naik (2013), the authors attempts to address the challenges of managing heterogeneous virtual environments to create a hybrid cloud platform. However, this proposed solutions involves proprietary technologies.

PaaSage is an interesting initiative for building a hybrid cloud solution from grounds and up using a defined deployment model, *Cloud Application*

Modeling and Execution Language (CAMEL) (PaaSage, n.d.; Zachariassen, 2015). PaaSage applications will specify tasks according to the CAMEL model, which will then be processed and then deployed on a platform according to the requirements. For tasks requiring high levels of SLA, PaaSage may determine to deploy the application on multiple cloud providers, including external cloud providers. PaaSage is a collaboration project, with contributions from large companies as well as research institutions. Lufthansa and Evry are are running prototypes and is among the main contributors of PaaSage and the project is scheduled to be finished September 2016. *Multicloud Deployment of Computing Clusters for Loosely Coupled MTC Applications* written by Moreno-Vozmediano, Montero, and Llorente (2011) published in an IEEE journal, explores the concept of deploying a computer cluster on top of a multicloud environment, but also in some configurations a hybrid type of setup, using both local hardware and rented hardware from Amazon EC2. The article investigates the viability of such a setup and analyzes the cost-performance ratio on each of the setups. According to the findings outlined in the paper, deploying a cluster on top of a multicloud environment scales linearly with little overhead. Additionally, in some hybrid setups, the cost-performance ratio was slightly improved compared to a pure local setup.

However, there are some aspects of the paper that does not fully translate to the practical issues a system administrator may encounter. The clustering technology used in this study was *Sun Grid Engine* (SGE), a long time veteran of approximately 15 years. Incidentally, Apache Mesos was designed to address some of the design weaknesses of SGE, in particular the use of static partitioning for the jobs run in the cluster, which prevents fully efficient utilization of the resources (Ghodsi, Hindman, Konwinski, & Zaharia, 2010).

OpenNebula is a cloud management technology with rich feature set to handle virtual machines and supports both hybrid cloud deployments and cloud bursting (OpenNebula Projec, n.d.). However, installation and management of OpenNebula is fairly complex as it consist of many parts to facilitate the rich feature set of a virtual machine manager. With virtualization of infrastructure, resources, network, and storage, there are many vectors for failure and SLA rates may also be be affected. Additionally, with clustering technologies like Apache Mesos, virtual machines may not be necessary and may even deter performance due to virtualization overhead.

According to a paper written by Iosup et al. (2011), public cloud providers does not perform as good as a local alternative for the purpose of *many-task computing* (MTC). This is mainly due to resource time sharing and the potential overhead virtualization may incur. However, the study was conducted around 2011 and may be obsolete, which the authors themselves also state in the conclusion. This is due to services like Amazon *High Performance Computing* (HPC) that has introduced since the article was published that seems to have addressed the issues outlined in the paper to considerable degree (Amazon Web Services, Inc., n.d.-b). Despite this, there may still be use cases where a private data center or cloud may be preferred for other reasons, like privacy concerns and regulations.

For the aspect of segmented workloads, a paper written by Jayaram et al. (2014) looks into the problems regarding geographical segmenting of hybrid clouds and discuss the challenges of such a mechanism. Several issues regarding trust management, attestation, and integrity management are analyzed in the paper. As the paper states, one of the main issues with a hybrid cloud is the trust issue regarding proper segmenting of the data. How can one be sure that data stored and processed in one area does not leak into an another?

Despite of the myriad of solutions and findings related to hybrid cloud within both the scientific and commercial communities, there has been no practical demonstration of using open-source and freely available clustering technology to attempt to address the multitude of challenges with creating a hybrid cloud platform that is available and supports data segmentation. This thesis outlines an attempt to prototype such a solution in addition to facilitation of cloud bursting, using spot price instances.

Chapter 3

Approach

This chapter will outline and explain the methods, processes, objectives, and general approach to solve the defined problem statement.

3.1 The objective

The objective for this thesis is outlined in the problem statement in the introduction chapter section 1.1:

How can we build highly available, segmented computer clusters using private computer hardware together with public cloud providers as a hybrid cloud platform, leveraging spot price instances for an automated cloud bursting solution?

The problem statement can be broken down into several sub-tasks that needs to be addressed for the whole question to be adequately answered.

Using the definitions written in the introduction chapter section 1.1:

- How can one build a computer cluster on top of private computer resources in addition to multiple public cloud providers.
- How can one, by utilizing both a private hardware and public cloud providers, gain improved levels of availability?
- How is it possible to segment data and data processing to specified locations or groups?
- How can one automate the use of spot price instances to accommodate for cloud bursting?

The main goal of this thesis is to prototype and implement a hybrid cloud solution that satisfies the requirements outlined in the problem statement. Therefore, the feasibility of deploying a hybrid cloud platform on multiple public cloud providers with increased availability, support for segmentation and facilitation of cloud bursting use cases is paramount in this thesis.

3.2 Formalization

As a large part of the thesis is about designing and evaluating system designs, it is important to describe those designs with accuracy and there are many ways to express that. A formal and detailed way of describing a system design is with the use *Unified Modelling Language* (UML). UML consist of various diagrams, each describing a specific aspect of the system in order to visualize the design of the system accurately. Used correctly, even complex systems can be accurately described with UML. However, with the increasing level of system complexity, so do the UML diagrams increase in both amount and complexity of the diagrams.

In this thesis, a subset of the UML diagrams available will be used to describe the design when appropriate, in addition to more general types of figures and text. The UML diagrams used will not necessarily be as according to the UML specifications and will be used solely to illustrate.

3.3 The testbed

The testbed will consist of the following main technologies:

- Cloud:
  - Altocloud - as the private cloud/data center
  - Amazon Web Services, *Virtual Private Clouds* (VPC) - as the public cloud provider
- Cluster:
  - Apache Mesos - as the clustering technology

3.3.1 Choice of technologies

Altocloud is the local OpenStack installation at Oslo and Akershus University College of Applied Sciences and is available for students and employees at the university college. For this thesis, Altocloud will emulate a private data center or cloud installation and will function as the baseline in the hybrid cloud configuration. In a production environment this would emulate an organizations in-house data center, installed and maintained specifically for the organizations private use. In addition to being able to boot up instances, Altocloud makes it possible to manipulate and manage the virtual network around the instances. This makes Altocloud a preferred choice over setting up physical machines, as it requires more time and can not be self-managed at the degree which the virtual environment Altocloud gives.

In regards of the choice of public cloud providers, several options were considered. DigitalOcean, Linode, Softlayer, Rackspace were a few of those considered, though in the end, the wide array of features available and the maturity of Amazon Web Services made it the final choice for the public cloud component of the hybrid cloud configuration. Amazon Web Services provides multiple geographical regions, each which for the purpose of this thesis can function as a separate public cloud provider for the hybrid cloud configuration. It is also the only public cloud provider that offers spot price instances. Additionally Amazon Web Services VPC allows quite extensive manipulation of the emulated network in the virtual private cloud.

For the choice of clustering technology, Apache Mesos has been chosen. Apache Mesos is a relatively new and emerging piece of technology which is fully open source and used in production environments in large companies like Twitter and AirBnB, attesting for the maturity and stability of the technology. Other technologies considered for the thesis include Kubernetes and Docker. While the mentioned technologies can to some degree be viewed as competing technologies, the mentioned technologies can be deployed in a single installation to leverage the advantages of each one.

3.3.2 Other considerations

As the focus of the thesis is resolves around Apache Mesos, hybrid cloud configuration and cloud bursting, the choice of other aspects of the solution will not be given much priority.

The choice of a operating system for running the Apache Mesos cluster is not vital for this thesis. Apache Mesos will run on most of the common and popular Linux distributions, and while the initial decision was to use CentOS, the final choice became Ubuntu LTS 14.04 x64. The choice was made on the basis that Mesosphere, a start-up company focusing on Apache Mesos and frameworks for the technology, provides a repository for this distribution which allows for easier install and maintenance. In addition, there are readily available Ubuntu LTS 14.04 x64 images on both Altocloud and Amazon Web Service EC2, with the recommended Ubuntu image at EC2 being eligible for the free tier, which allows for free t2.tiny instances at Amazon Web Service EC2.

Other aspects of the testbed that needs to be considered is the instance type of the virtual machines running the different components of the cluster. As shown in Table 3.1, the instance types at Altocloud and Amazon Web Service EC2 are not identical and while there are some similarities, the instance types deviates enough to warrant caution when setting up a hybrid cloud platform, especially considering the number of vCPUs. Do note that only a subset of the currently available instance types at Altocloud and Amazon Web Services EC2 are listed up in Table 3.1 and that Amazon Web Services EC2 also offers specialized instance types for specific types workloads.

	Altocloud	**AWS EC2**
	m1.small	**t2.small**
vCPUs:	1	1
RAM:	2048 MB	2 GiB
	m1.medium	**m3.medium**
vCPUs:	2	1
RAM:	4096 MB	3.75 GiB
	m1.large	**m3.large**
vCPUs:	4	2
RAM:	8192 MB	7.5 GiB

Table 3.1: Specifications for subset of instance types available at the Altocloud and Amazon Web Service EC2.

The instance types has to be carefully considered, more ideally chosen based on benchmarks for the specific use. While this does not have a large consequence regarding the Mesos slave nodes, as they can flexibly be added, modified and removed depending on the workload, the backup Mesos master nodes needs to be dimensioned with consideration, as they are largely redundant and does not necessarily actively participate in the workload of the cluster, with the exception of keeping redundant states in case of a master node failure. For a small cluster with little network activity, a small instance type for the master node may suffice. On the other hand. with a large cluster consisting of 1000+ slave nodes, it would require a larger instance type to handle the workload. For small clusters, the master nodes can in addition to the Mesos master process, also run a Mesos slave process, thus participate in the cluster as slave node to process tasks.

3.4 Outlining the design

A considerable part of the thesis will be to develop, model and evaluate different designs of setting up a Apache Mesos cluster. As previously explained in the background chapter 2, Apache Mesos clusters can be very roughly be described as a master-slave architecture consisting of *master* nodes and *slaves* nodes.

At all times, only one master node is active, with any additional master nodes functioning as live backups in case the active master node should become unresponsive. Depending on the SLA requirements, there may not be need for any redundant master nodes at all, or it may require five or more master nodes due to higher SLA requirements. However, as the number of master servers increases, the complexity of electing the active master node increases. The number of master nodes should not be arbitrary and should be carefully considered, as this is one of the few parts of a Apache Mesos cluster that results in almost fully idle resources as backup.

Another important aspect that needs to be considered regarding the master nodes are their distribution. How can one distribute the master nodes in a hybrid cloud architecture with respect to certain requirements? In the case of *high availability*, the most obvious and immediate solution would be to spread the master nodes to different locations to minimize the risk for the platform as a whole to fail. However, if *privacy* is of utmost importance, then that would require the master nodes to be placed at locations that fulfill the privacy requirements, which may very well be only a single location. There are also other aspects like price, latency, and throughput requirements that would require different setups to fulfill.

As for the cloud bursting scenarios, it is important to consider the use cases for such a setup and how it fits with existing requirements like high availability or segmentation of the data. Cloud bursting by itself may require a specific hybrid cloud configuration for it to function optimally.

The problem statement considers three aspects of a hybrid cloud solution:

- high availability
- segmentation of the data
- cloud bursting

As a result at least two main design models will be proposed and implemented that considers high availability and segmentation of the data. Additionally the solution for automating cloud bursting with the design models proposed as a viable option will be considered.

3.5 Prototype implementation

One or several prototypes will be implemented according to the proposed models defined in the design phase to verify the feasibility of the configurations and to test the models against a set of criteria deviated from the problem statement.

3.5.1 Verifying the implementations

To verify that the prototypes fulfills the requirements of the problem statement several test scenarios will be tested.

Availability

To accurately measure availability in a real life scenario, one would need to sample the availability over a long period of time, with the at least a year as a baseline to even be able to make somewhat accurate assumptions. Due to the time constraints for this thesis, this is not possible. While statistical analysis is powerful, even with a small sample, due to the relatively low failure rates of cloud services, there may very well be a rate of 100% availability within period of a few months, thus rendering statistical analysis powerless. As a result, theoretical simulations and calculations will be conducted and should for the purpose of this thesis be adequate for answering the problem statement.

Availability has different prioritization and aspects that needs to be considered. In particular the amount of resources and complexity. There has to be a degree of efficiency involved, as redundant, but a passive resource pool is not very efficient. Additionally, it is important to consider for who the availability is important for.

The test scenarios that will be simulated are as follows:

- A Mesos slave process becomes unavailable.
- The working Mesos master process becomes unavailable.
- An entire region within the hybrid cloud becomes unavailable.
- The hybrid cloud splits and semi-isolates part of the platform.

A Mesos slave process becomes unavailable
In this scenario, a Mesos slave process on becomes unavailable. There are numerous possible causes for this to happen, including hardware failures, network troubles, failures in operating system or kernel, and problems with the Mesos process itself.

The working Mesos master instance cease to function
Similar to the previous scenario, this scenario covers the event of a Mesos master becoming unavailable for some reason.

An entire region within the hybrid cloud becomes unavailable
This scenario covers the possibility of an entire region failing within the hybrid cloud. This could the entire private data center or it could be that a public cloud provider is unavailable due to networking problems. In the case where multiple availability regions are used, each one of them will be considered a region within the hybrid cloud platform.

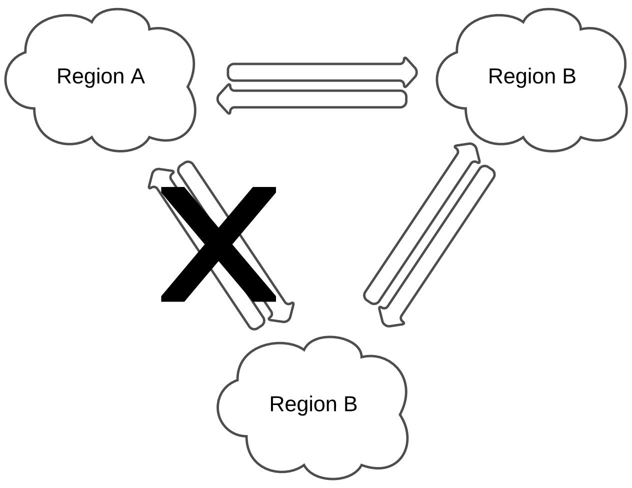


Figure 3.1: Region A can not reach Region B, but Region B can.

The hybrid cloud splits and semi-isolates part of the platform
This covers the scenario of part of the the hybrid cloud becoming split, possibly due to a split in the network. This could result in parts of the hybrid cloud having degraded reachability, thus resulting in a split or semi-isolated network. This scenario is illustrated in Figure 3.1.

Segmentation

Verification of segmentation of tasks and data will be done by confirming that segmented tasks are segmented as specified. For the aspect of privacy, it is difficult to really be sure that traffic is truly private and does not at any time leak out. To verify that that no data regarding the tasks are leaked to undesired locations, the source code of ZooKeeper, Apache Mesos and the used frameworks has to be inspected. For the time constraint of this thesis, this is not possible. To narrow down the scope of this project, a major assumption has been made for the sake of analyzing the aspect of privacy and segmenation.

The flow of the internal traffic of Apache Mesos has been briefly covered in the Background chapter 2. Initially, only traffic regarding resource availability at the Mesos slave node is sent to the Master in addition to traffic related to the cluster itself and keep-alive pings is assumed to be sent at a regular interval. It is therefore assumed, based on the official documentation of the internal traffic that no information about the available tasks are sent to the Mesos slave nodes before the tasks are being granted (Hindman et al., 2011). This means that only when tasks are being handed out after checking any constraints, will there be any other traffic than the traffic necessary for the cluster to function. Consequently, information about the tasks are sent out on a need-to-know basis. An UML sequence diagram is shown in Figure 3.2 and describes the a possible interaction between the different Apache Mesos components, assuming *MasterY* is the working Mesos master node.

Note that the sequence diagram is simplified and not meant to be fully accurate. It is included to illustrate the a possible interaction between the components.

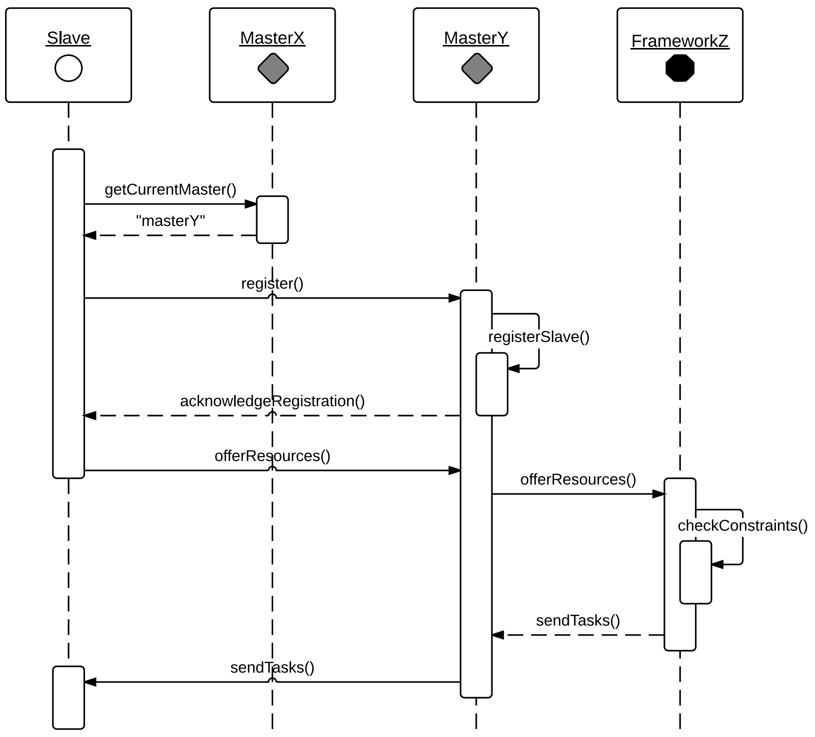


Figure 3.2: An UML Sequence diagram describing a possible interaction sequence assuming *MasterY* is the working Mesos master node.

Automated cloud bursting

To build a cloud bursting solution there are several obstacles that needs to be cleared. In traditional data centers the first major obstacle to cloud bursting is the lack of integration. There are no easy way to offload or expand workloads into public cloud platforms, let alone other platforms at all. For cloud bursting to be a feasible option, the solution has to be simple, easy, and fast enough to be useful. As a result the solution will have to utilize cloud bursting as efficiently as possible and should be able to allow applications and processes to burst into the cloud with little to no interference with existing services and no downtime.

It is possible to offload workloads into the public cloud, using more traditional means like virtual machine migrations. However, this is a fairly rigid and lengthy process, especially over the Internet, which for some use cases can result in unavoidable downtime. Furthermore, a simple migration may break dependencies and there might simply not be any straightforward way to burst into the cloud from the existing environment. In addition, the time and resources spent on migration may negate the potential benefits of offloading the workload to the cloud, possibly due to the sheer amount of resources needed to migrate or due to the time span for which the extra resources are needed.

A cluster on a hybrid cloud solution will solve this by pooling together physical and virtual resources and abstracting it. Frameworks running on top of Apache Mesos will only see a pool of available resources, regardless of the location of the the actual resources.

The solution should be able to:

- start and stop cloud instances
- collect at least one variable to aid in the decision making
- based on the variables collected, decide whether or not to burst into the cloud
- leverage the use of spot price instances

In addition, experiments will be conducted to simulate bursty workloads to verify the functionality of the cloud bursting solution. The simulations will cover two aspects of a cloud bursting scenario with increasing and decreasing workloads. The automatic cloud bursting solution should, when needed, automatically burst into the cloud and then maintain and scale the number of Mesos slave nodes located at the public cloud site.

3.6 Considerations and limitations

There are several known limitations to the thesis and the technologies at hand that will needs to be considered in a real-life scenario.

The thesis assumes the use of Apache Mesos as a clustering technology and derives the possibilities and design based on the functionality provided by the Apache Mesos as a piece of technology. As a result, any limitations to Apache Mesos will also be true for the final solution of the thesis.

A potentially large limitation which has to be considered before deploying an Apache Mesos cluster is that it runs a Linux environment. This means applications that needs to be run on other types of operating systems like Microsoft Windows, will not function.

Additionally, as mentioned in the background chapter 2, there are currently no support for encryption of the internal traffic between the master and the slave nodes in the current version of Apache Mesos. This is especially severe if the traffic is going out on the Internet. Measures should be taken to secure the traffic using other means like *Virtual Private Network* (VPN) tunneling if an Apache Mesos cluster is to be deployed in a production environment.

By that extension, it is assumed that Apache Mesos is intended to be used on private networks, which is also evident in how it benefits from fast networking with low latency, assumably to keep the overhead of node management low (alexr_, 2015). Consequently, deploying Apache Mesos in a hybrid cloud configuration could potentially result in unoptimal performance and sudden timeouts as a result of the higher latency incurred by the hybrid cloud configuration. However, in a paper written by (Bicer, Chiu, & Agrawal, 2011), the authors present a distributed system with the overhead incurred by remote retrieval and potential load imbalance amounting to approximately 15%. Furthermore, the authors argue that the overhead is at a manageable level which makes cloud bursting feasible and scalable for the distributed systems they were testing. Their findings could possibly also be true for Apache Mesos if the configurations are tuned for a hybrid cloud setup.

As mentioned in the Background chapter 2, Apache Mesos is currently not designed for the use on the Internet and it is therefore not a given that the implementation of the proposed designs will work. Consequently, additional solutions will be considered in the event that Apache Mesos does not work as intended outlined in the designs.

A known limitation to the testbed is the lack of a working domain in the network. This means that local hostnames in the network will not be resolved. The installation and maintenance of *Domain Name Server* (DNS) is outside the scope of problem statement and will not be included. Instead, IP-addresses will be used directly. Due to this, any peculiar glitch related to DNS will be mitigated should they arise. In a production environment, a proper domain, even a local one, should be set up with a DNS properly resolving it.

3.7 Expected results

For the design and implementation of the highly available cluster, several designs will be proposed and implemented. The prototypes will then undergo testing as outlined in the test scenarios to analyze the behaviour and the characteristics of the prototypes. These prototypes will then be discussed with the problem statement in mind in addition to potential value within the field.

As for the design and implementation of the segmented cluster, as long as the assumption regarding the flow of traffic within Apache Mesos holds, the design and implementation should be successful. With the assumption withstanding, the use cases for such a system will be analyzed and discussed.

Regarding the automated cloud burst solution, it is expected that a rudimentary solution will be prototyped and tested. The cloud bursting solution is expected to fulfill the requirements stated, however, the prototype will most likely not include any extra functionality, and may contain smaller bugs. The solution will then be analyzed and evaluated against the problem statement.

Chapter 4

Results: Design

4.1 Overview

This chapter will describe the proposed solutions for addressing the problem statement, discovered limitations and additional considerations regarding the set up of a Apache Mesos cluster in a hybrid cloud setup. Additionally, a solution for automating cloud bursting, leveraging spot price instances to maximize the cost effectiveness of the proposed solution will be proposed.

The proposed designs and solutions in this chapter will be tailored to the testbed environment described in the Approach chapter 3. Further details, adjustments and specifications regarding the testbed will described in this chapter.

4.2 Environment

As alluded to in the Approach chapter 3, the choice of instance type for the Mesos master node has to be given extra consideration, as it is the only part of the cluster which results in almost fully idle resources. In a paper written by Hindman et al. (2011), a series of performance benchmarks were conducted to gauge the capacity of certain parts of the Apache Mesos cluster. With a 8 vCPUs and 6 GB RAM instance at Amazon Web Services EC2 used as the Mesos master node, the scheduling of tasks and internal processing required for the cluster adds an overhead of less than one second, with 50 000 Mesos slave node in the cluster. The paper was published in 2011 and it is assumed that further performance optimizations has been done since then, reducing the overhead even further.

For this project, a medium instance type will be used for the Mesos master nodes. For Altocloud this would be *m1.medium*, while at EC2 it would be *m3.medium*. While a small instance type may have worked out, a medium instance type was chosen. This was done as a safety measure, as too

	Altocloud	**AWS EC2**
	m1.medium	**m3.medium**
vCPUs:	2	1
RAM:	4096 MB	3.75 GiB

Table 4.1: The specifications for the instance types chosen for running Mesos master nodes.

low performance might result in unexpected glitches, especially regarding network performance. In order to be sure this would not cause an issue, the instance type was over-dimensioned by a considerable margin. The focus is to evaluate and design the cluster for high availability, segmentation of date in particular and cloud bursting scenarios. Performance as factor is less of a priority. The specifications are for the medium instance types has been re-listed for convenience in Table 4.1. It should be noted that it is common to run Mesos-frameworks on the same machines as the Mesos master nodes, and the machines should be dimensioned for this extra load, unless the frameworks are running on separate machines.

Since it was determined that Amazon Web Services would be the public cloud provider for the hybrid cloud setup, the availability region has to be determined. When deciding the availability region to be used, it is important to consider price, latency, and regulations among other things. Due to the time frame for the project, the overall price difference will be minor. The latency to the availability region was therefore given priority. As a result, the availability regions EU Central (Frankfurt) was picked to be the primary choice, with EU West (Ireland) as the secondary choice.

4.3 Architecture

This section will contain the proposed solutions for addressing the problem statement in three main parts concerning availability, segmentation of data and automated cloud bursting.

4.3.1 Availability

Depending on the the level of availability needed, it is necessary to consider fault-tolerance at multiple levels from the fundamental level hardware resources up to the individual applications. A key technique for improving availability, is to duplicate and keep a redundant copy or backup of the entity one wish to improve. This could mean physical machines, network links, power circuits, or multiple instances of a application. In most cases, it is also possible to use the redundancy to load balance, which lowers the overhead of keeping a duplicate.

However, it is imperative that the redundant entities are as independent from each other as possible in order minimize the risk for failure. Consider the following scenario: A service is running on two servers located in a single data center and load balancing distributes the load between them. While the the servers are separate, they both depend on the data center being functional, which may experience full blackout due a disaster of some type. In order to improve availability, the servers should ideally be placed in two separate data centers.

To achieve higher levels of availability, Apache Mesos can be deployed on multiple cloud platforms and data centers, forming a hybrid cloud. Before setting up a Apache Mesos cluster, several things has to be considered.

In a Apache Mesos cluster, the Mesos master node is responsible for managing the cluster and is therefore vital for the cluster. The number of master nodes in a cluster must be odd. This is required in order to prevent a split brain problem, which occurs when the cluster splits into two or more smaller clusters, each with their own Master node managing the cluster. This is a problem as it results in inconsistent states for the cluster as a whole and must be prevented. With an odd number of Mesos master nodes correctly configured, more than half of the Mesos master nodes needs to be able to communicate to change the state of the cluster.

Prototype 1: Maximizing availability

Figure 4.1 illustrates a solution using three separate cloud platforms as components of the hybrid cloud. The Mesos master nodes are distributed between the availability zones, with a few Mesos slave nodes arbitrarily present in each of the zones. The failure rate of each of these zones are assumed to highly independent of each other, which will lower the the risk of failures of the the underlying resources used by the Apache Mesos cluster.

For this prototype, the officially recommended number of three Mesos master nodes has been chosen. For a proof of concept, this will suffice. Should the underlying resources the Mesos master nodes utilize have a high failure rate, a higher number of Mesos master nodes should be considered.

This setup will maximize availability, which is ideal for public facing services that does not have any particular requirement other than to maximize availability. A public service like a website, back-end infrastructure for advertising services or smartphone applications are examples of such types of services.

Due to the architecture of Apache Mesos, any slave node can become unavailable without affecting the overall service availability. As long as the framework deploying the services is fault-tolerant and there are enough slave nodes to be able to accommodate for the bare minimum of

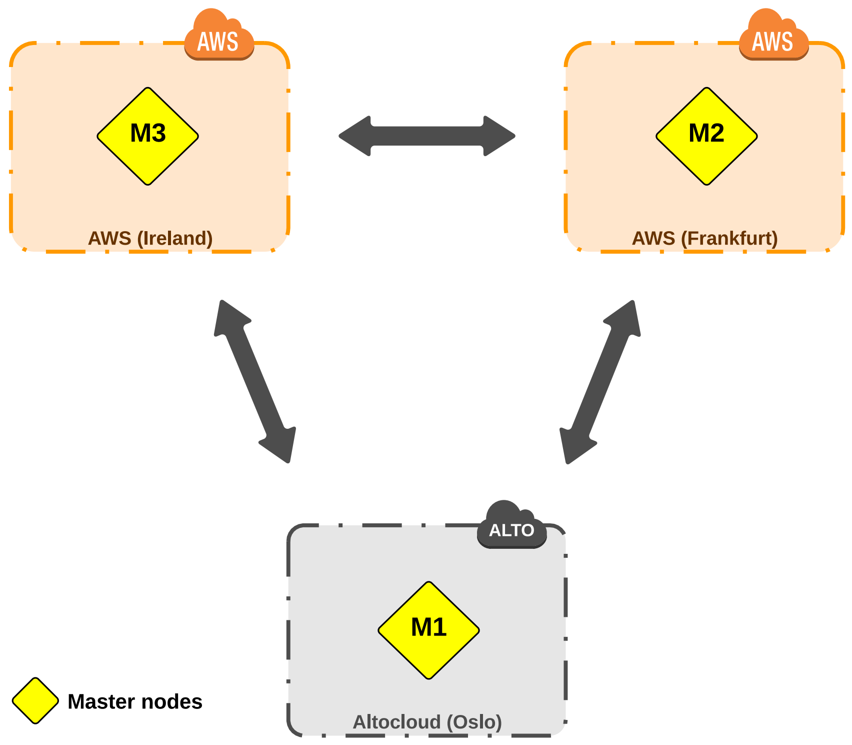


Figure 4.1: A hybrid cloud setup, distributing the Mesos master nodes to independent availability zones.

the processing needed to keep the services running, a large amount of slaves can fail simultaneously. Additionally, this setup accommodates for downtime in an entire availability zone, the hybrid cloud platform will still be functional in that scenario.

This setup requires each and every node of the network to have a public IP-address which is routable on the Internet. This is a huge limitation, as the number of IPv4 addresses are limited. Alternatively, the number of Mesos slaves nodes deployed on Altocloud can be reduced or removed altogether, relying on the public routable IP-addresses Amazon Web Services EC2 provides each node by default. Another solution is to set up VPN tunnels between the availability zones and route with private addresses.

Prototype 2: Prioritize local availability

This prototype is illustrated in Figure 4.2. In this suggested setup, local availability is given a priority, with the majority of the Mesos master nodes present in the local cloud. This particular setup uses five master nodes, but can easily be extended to a higher number as long as the majority of the Mesos master nodes are located in the local cloud.

Even if the access to the Internet is lost, local access can be gained from the

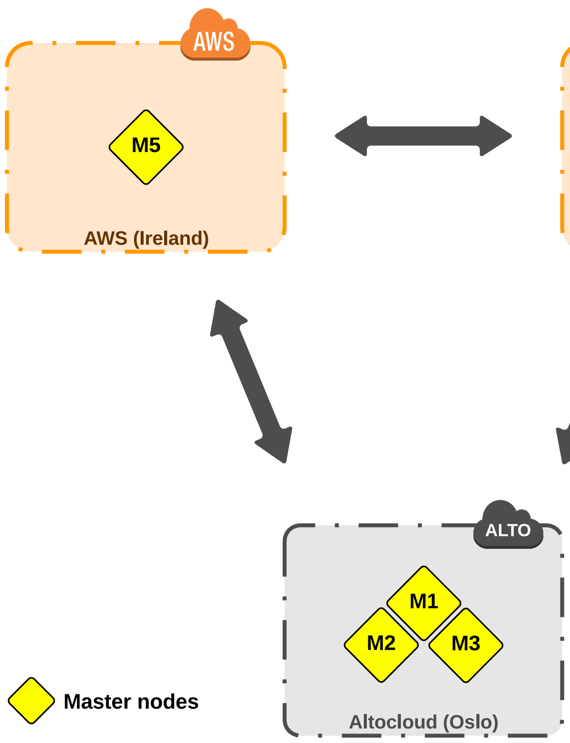


Figure 4.2: A five Mesos master node cluster with the majority of them located at Altocloud.

same network as the local cloud and the services running on the cloud will continue to be responsive for internal use. This makes this type of setup ideal for local services like *Enterprise resource planning* (ERP) applications, local batch jobs, and analysis frameworks like Hadoop running, which is not normally publicly accessible. The proposed setup will function well where the private cloud or data center is used as the preferred baseline location, with public cloud resources added to increase processing power and/or availability.

However, this setup does not account for the opposite scenario and depends on a the stability of the private site to keep the cluster running. While the master nodes themselves are independent, in the event that the entire private data center goes down, the entire cluster will experience downtime, regardless of the total number of masters. This is due to the fact that for the cluster to remain available locally should the site be isolated, the site needs to have the majority of the Mesos master node present, which effectively prevents the hybrid cloud platform from functioning without the private site.

This setup does not give the highest availability possible, but sacrifices the overall level of availability for the priority of local access. This prototype works well in scenarios where cloud bursting is desired, as the private locations functions as a baseline platform that scale out on public cloud platforms.

Like the first prototype, this setup requires an IP-address for each node in the cluster. As this is not feasible for larger clusters, setting up VPN gateways and tunnels between the sites will abstract the network, providing a virtual private network with a numerous of private IPv4 IP-addresses.

Fallback solution: VPN tunneling

A fallback solution has been prepared, in order to account for the possibility that Apache Mesos does not work as intended for the proposed prototypes or any other networking issue that may cause an issue. In those cases, an instance dedicated to work as a VPN gateway will be set up at each site, which will establish VPN tunnels between the them. This can be done due to the amount of control Amazon Web Services gives over the network with their VPC service. Likewise, Altocloud, being an OpenStack installation, also permits a higher level of manipulation of the network.

IPSECv2 tunnels will be set up at VPN gateways and will be installed and configured through Openswan, an IPSEC implementation that supports an array of features, with NAT traversal being particular interesting. As mentioned previously, Apache Mesos currently is developed and maintained to work in private networks configurations and with the way public IPs are handled by the cloud platforms, NAT traversal may end up being the main issue.

Correctly set up, a VPN environment will for the machines present in the network function just like any other private network.

4.3.2 Segmentation of data

The segmentation of the hybrid cloud can be facilitated by existing functionality in Apache Mesos and the Marathon framework. Every node in an Apache Mesos cluster can be tagged with a arbitrary number of attributes, which will be sent along with the resource offers from a Mesos slave node. The frameworks can by reading the attributes in combination with the offered resources to make a decision of whether or not to accept the offered resources.

There are, in addition to Marathon, other Mesos frameworks that manages long-running services like Aurora and Singularity. However, Marathon was picked due to the simple graphical interface, well documented REST API, and a clear cut way of enforcing constrains, which is vital for segmenting the hybrid cloud.

Marathon supports a number of operators to set constraints on how applications are run, with *CLUSTER* being the operator of interest. This operator will require the tasks to run in a cluster constrained by the attribute defined. For constraining an application to only run on nodes where the attribute *color* is set to *blue* then a constraint will look like: *color:CLUSTER:blue*.

For segmenting the data, an attribute will be set on every slave node in the Mesos cluster. The attribute will be named *cloud_type* and contain either *public* or *private*. Attributes can be set when starting the Mesos services or pre-configured in files. Another way to segment the cluster is to create an attribute named *clearance_level* and have values that mimic the clearance levels of public, confidential, secret, and top secret.

4.3.3 Automated cloud bursting

A cloud bursting solution with a hybrid cloud setup utilizing Apache Mesos to provide an abstraction layer on top of physical or virtual resources makes cloud bursting as simple as adding a Mesos slave node located in the cloud. The slave node will join the cluster and offer its resources within minutes after booting up and start processing tasks.

The challenge therefore becomes to automate it and to utilize the spot price instances, while considering the billing-cycle interval of one hour and price fluctuations. A prototype will be written in Python and will run on top of Marathon in a fault-tolerant manner.

The script will consist of three main parts:

- Data collection
- Price and scaling decision logic
- Management of the spot requests and instances

The first part of the script is the data collection and encompasses the collection of metrics from Apache Mesos and through the Amazon Web Services API that will aid in the decision-making process. Interesting metrics includes total resources available, resources in use, spot requests pending, number of spot instances, and meta data connected to each of the spot instances.

Using the collected information, a decision of whether to scale up or down the cloud bursting capacity is made. Depending on how many factors and considerations one wish to account for, this step is potentially highly complex.

The paper written by Voorsluys and Buyya (2012), briefly mentioned in Background chapter 2, describes five bidding strategies that can be used. The bidding strategies are listed up in Table 4.2, where $G = 0.001$, which is the lowest granularity value allowed to be used at Amazon Web Services when bidding for spot instances.

Bidding strategy	Bid value definition
Minimum	The minimum value observed in the price history + G
Mean	The mean of all values in the price history
On-demand	The listed on demand price
High	A value much greater than any price observed
Current	The current spot price + G

Table 4.2: List of five possible bidding strategies for the spot price instances at Amazon Web Services EC2 (Voorsluys & Buyya, 2012).

As there are several aspects of the proposed algorithms in the mentioned paper that does not apply directly to the hybrid cloud scenario, a simpler decision algorithm has been devised, which is adequate for the purpose of answering the problem statement. In Figure 4.3, the projected activity flow is illustrated.

At the launch of the script, the values of available resources at the private location will be provided through a configuration file. Based on the total available baseline resources and the resources currently in use, a percentage is calculated. This percentage will be the sole variable which will determine whether or not to utilize public cloud resources. In a live production a more complex decision algorithm should be considered, as the one proposed for this prototype does not account for any edge cases.

The script will operate with a *burst point threshold*, which represents the usage-percentage of the available resources at which the cloud should scale up. For instance, if the *burst point threshold* is set to *0.70*, the script will request spot instances at when resource usage reaches 70%. Additionally, a specific maximum limit will be artificially set in order to prevent the prototype from scaling up too much and incurring huge costs.

The decision to scale down will be based on actual usage of the resources and the lifetime of a spot instance. Lifetime has to be considered, due to to the hourly billing cycle that applies when the termination of an instance is initiated by the user. To avoid wasting already charged resources, a spot instance will be terminated only if the lifetime of the instance is nearing the next hourly billing-cycle. For example, a spot instance may only be needed for 20 minutes, but will not be user-terminated for at least 30-35 more minutes to capitalize on the already charged hour.

The chosen bidding strategy is the *current price* + x, where x is a predefined variable that is supplied with the script at launch time. x serves as price padding to avoid bidding at the exact market price, bids at that level will be highly contested. By setting x to a high value, one can increase the likelihood for the instance to last longer before terminated. On the other hand, by setting a low value, the instances could potentially be terminated due to increased prices, which would result in the charge for the last partial hour to be waived. In the end, the script will bid the lowest price possible plus a padding value, which can be arbitrary set to tweak the bidding strategy.

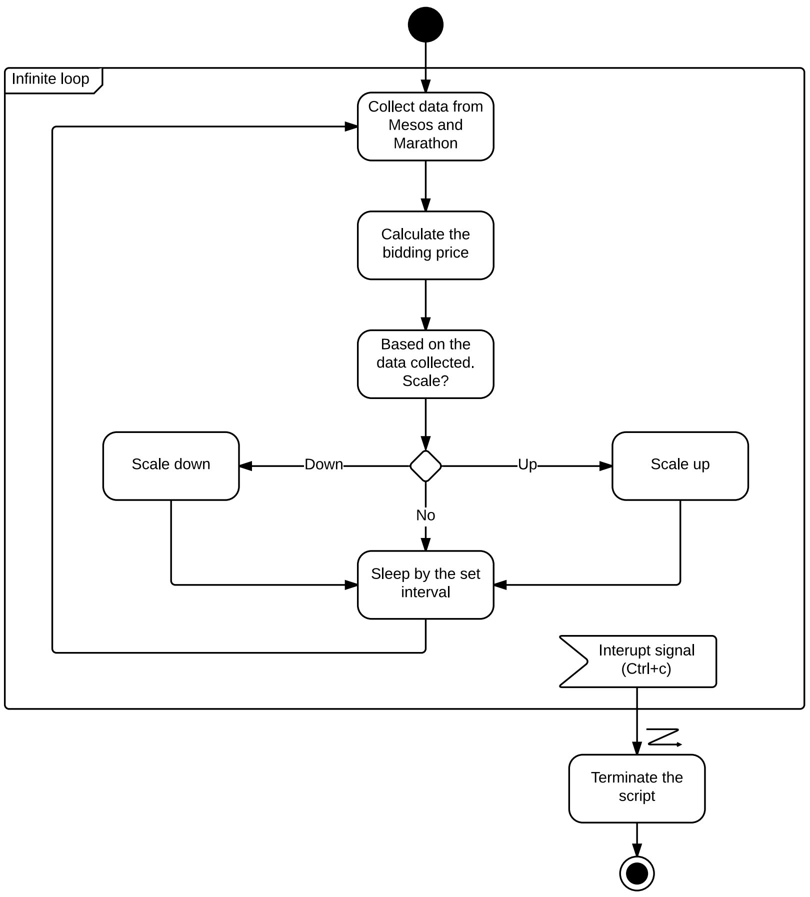

Infinite loop
Collect data from Mesos and Marathon
Calculate the bidding price
Based on the data collected. Scale?
Scale down
Down
Up
Scale up
No
Sleep by the set interval
Interupt signal (Ctrl+c)
Terminate the script

Figure 4.3: Projected activity flow for the script illustrated in an activity diagram.

Chapter 5

Results: Implementation

This chapter describes the actual implementations of the proposed prototypes and solutions, the testbed, and other aspects that is relevant for addressing the problem statement.

5.1 Setting up the testbed and Apache Mesos

As described in the Approach chapter 3, the testbed uses Altocloud and Amazon Web Services VPC to emulate a private site and public cloud platforms, with the Apache Mesos chosen as the clustering technology.

There are several ways of installing and managing Apache Mesos. Configuration management tools can be used to bootstrap and manage the Apache Mesos binaries in addition to other miscellaneous configuration. However, in order to to have full control over the installation process, the Apache Mesos was installed manually. This is due to the deployment prototypes being non-traditional and of experimental nature. In contrast of modifying and tweaking existing configuration management templates, a manual installation gives more control over the installation process which eases the debugging process.

During the installation of Apache Mesos master nodes, several issues were encountered. One of the issues caused the Mesos master nodes to not reach full equilibrium, resulting in a new leader being elected every minute, flip-flopping between the master nodes present. This was found to be a DNS related issue and was mitigated through an entry in the `/etc/hosts` file, which effectively functions as a manually maintained and makeshift DNS entry. Additionally, due to a bug in ZooKeeper, an Apache Mesos process will crash if proper DNS handling is not in place (Apache Software Foundation, n.d.-b, n.d.-a).

An Apache Mesos cluster including both master nodes and slaves nodes were successfully installed and configured in Altocloud, with slave nodes correctly registering themselves to the cluster through the leading master node. However, when attempting to register a slave node running at Amazon Web Services EC2 peculiar activity was observed. The traffic from the slave node located at EC2 managed to successfully send a registration request to the leading master node, passing through multiple layers of network abstraction including two layers of NAT. Although the master node receives the registration requests, no registration acknowledge is ever sent back.

Eventually, the cause was discovered to be a combination of the use of NAT and the way Mesos nodes communicates between each other. When a slave node sends a registration request, it includes information about the resources available and an IP-address. The IP-address sent along is the one that is defined on the network interface bound by the Apache Mesos process. Furthermore, in a cloud environment like Altocloud and Amazon Web Services EC2, the public IP-addresses are loosely coupled with the virtual machine and functions similarly as NAT does. Consequently, the Mesos master attempts to send the acknowledgement and other internal traffic meant for that slave node to the non-routable private IP-address. The communication flow is illustrated in Figure 5.1.

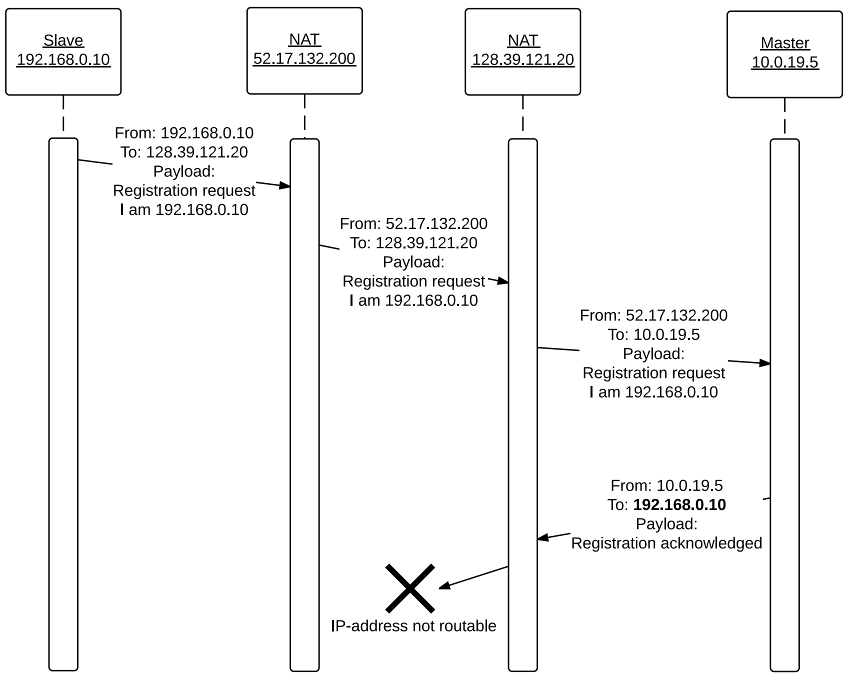

Slave 192.168.0.10
NAT 52.17.132.200
NAT 128.39.121.20
Master 10.0.19.5
From: 192.168.0.10 To: 128.39.121.20 Payload: Registration request I am 192.168.0.10
From: 52.17.132.200 To: 128.39.121.20 Payload: Registration request I am 192.168.0.10
From: 52.17.132.200 To: 10.0.19.5 Payload: Registration request I am 192.168.0.10
From: 10.0.19.5 To: 192.168.0.10 Payload: Registration acknowledged
IP-address not routable

Figure 5.1: Communication flow between an Apache Mesos slave node and master node with the registration attempt failing due to how public IP-addresses are handled in cloud platforms.

As previously indicated in the Results: Design chapter 4, NAT traversal appears to be an issue for Apache Mesos. In order to mitigate this limitation, a VPN solution has been deployed as outlined in the previous chapter. As the set up of a VPN solution is outside the scope of this thesis, it will not be explained in detail. The subnets of the VPN network was divided into three main parts for simplicity, using all three private address spaces as defined by RFC1918. The resulting network has been listed in Table 5.1.

Availability region / Site	Subnet
Altocloud (Oslo)	10.0.19.0/24
AWS (Ireland)	172.16.0.0/16
AWS (Frankfurt)	192.168.0.0/16

Table 5.1: The network partitioning of the RFC1918 private addresses divided into separate subnets.

A good chunk of the `172.16.0.0/12-subnet` has been left unallocated in case a need for additional private addresses arises.

5.2 Availability

5.2.1 Prototype 1: Maximizing availability

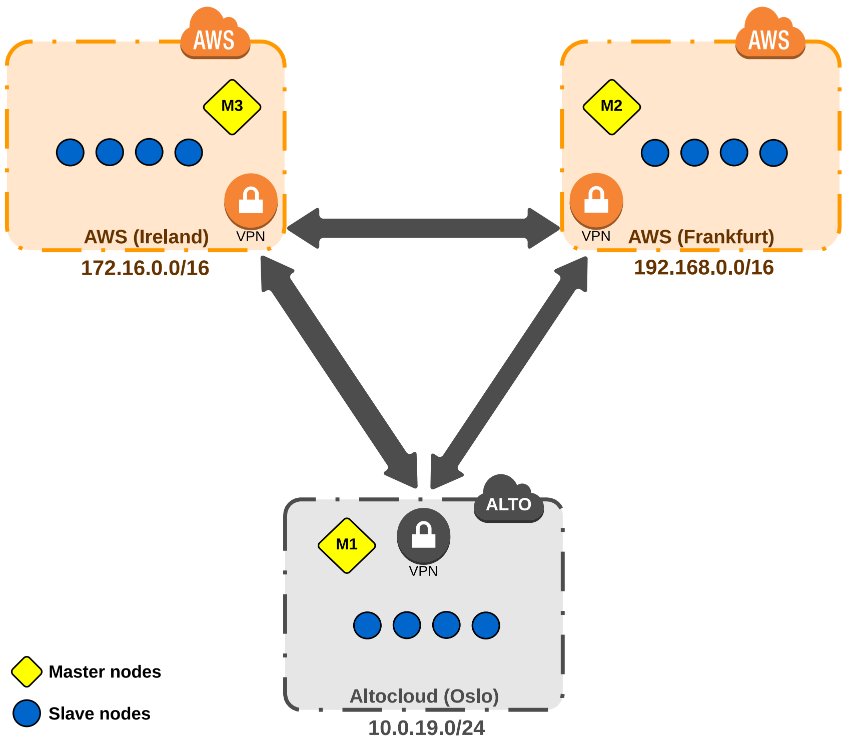


Figure 5.2: Prototype 1: Maximizing availability. Distributing the master nodes and thereby the risks.

By using VPN tunneling, the need for allocating public IP-addresses for each node disappears for the purpose of maintaining the cluster, as the private IP-addresses becomes routable within the hybrid cloud platform. With the exception of the extra infrastructure to maintain a VPN, the prototype is identical to the proposed proposed design. Figure 5.2 illustrates the final implementation of the prototype, showing how the Mesos master nodes are distributed between the different availability regions.

Slave ID	Host	CPUs	Mem	Disk
...5050-5669-S0	192.168.187.205	1	496 MB	3.9 GB
...5050-5669-S2	192.168.178.239	1	2.7 GB	3.9 GB
...5050-5669-S1	172.16.231.155	1	496 MB	3.9 GB
...5050-900-S71	10.0.19.9	4	6.8 GB	73.7 GB
...5050-900-S69	10.0.19.8	4	6.8 GB	73.7 GB

Table 5.2: A subset of available slave nodes in this scenario. The information has been taken from the Apache Mesos GUI and represents a truncated view of the available slaves.

Table 5.2 shows subset of available slaves that is listed up in the *graphical user interface* (GUI) of Apache Mesos. Note the subnet differences of the slaves, which indicates the where the slaves are located.

As for the Mesos master nodes, they were installed manually on top of instances booted up in their respective availability zones. The installation notes for setting up the Apache Mesos master nodes has been included as Appendix A. The specifications for the Mesos master nodes is listed in Table 5.3.

Master ID	IP-address	Location	Instance type
1	10.0.19.5	Altocloud (Oslo)	m1.medium
2	192.168.0.5	AWS (Frankfurt)	m3.medium
3	172.16.0.5	AWS (Ireland)	m3.medium

Table 5.3: A list of specifications regarding the Mesos Master nodes.

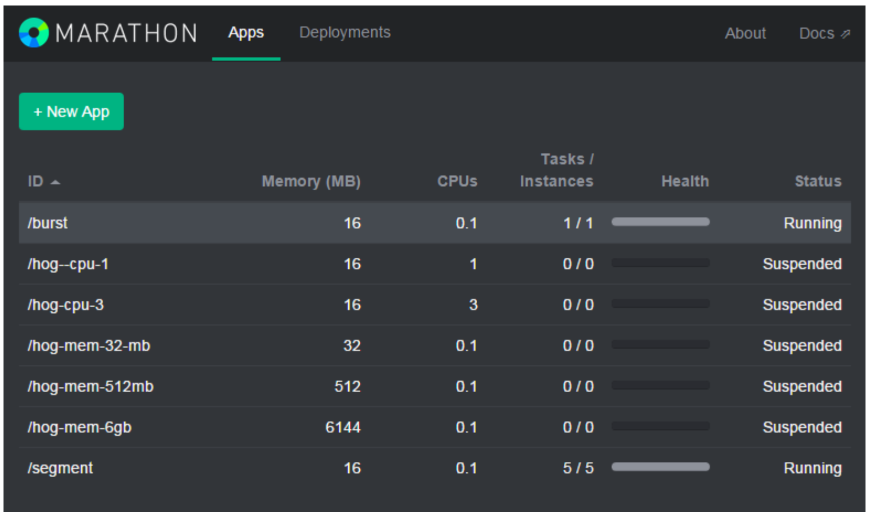
The Mesos slave nodes are set up using a bash script which is supplied at instance start-up as user-data. The script is included as an Appendix B

For creating tasks for testing the availability, the meta framework Marathon has been installed. Marathon exposes a small subset of the functionality of the Marathon REST API, through a very simple GUI. Although simple, it suffices for creating, destroying and scaling tasks. A screenshot of the GUI is shown in Figure 5.3

Test scenarios

A Mesos slave process becomes unavailable

In the event of a Mesos slave node becoming unavailable for some reason, the Mesos master node allows a default timeout period of 75 seconds to



ID	Memory (MB)	CPUs	Tasks / Instances	Health	Status
/burst	16	0.1	1 / 1		Running
/hog--cpu-1	16	1	0 / 0		Suspended
/hog-cpu-3	16	3	0 / 0		Suspended
/hog-mem-32-mb	32	0.1	0 / 0		Suspended
/hog-mem-512mb	512	0.1	0 / 0		Suspended
/hog-mem-6gb	6144	0.1	0 / 0		Suspended
/segment	16	0.1	5 / 5		Running

Figure 5.3: A screenshot of the Marathon GUI with a some created tasks available to scale up or down.

pass before procedures for deactivating the slave node is begun. Should the slave node start responding within this timeout period, nothing will happen and both the Mesos master node and the slave node simply ignores the temporary unavailability.

However, if the timeout period is exceeded and the slave nodes is still unavailable, the Mesos master node will attempt to deactivate the Mesos slave process on the slave node before it from the list of available slave nodes. Tasks that were lost will be rescheduled to other slave nodes with available capacity. In Listing 1 an excerpt of the Mesos master log is included. The events logged are the result of a simple reboot of the instance for this particular Mesos slave node.

```
17:00:26.087030 Disconnecting slave ...5050-5669-S0 at
 ↪  slave(1)@192.168.187.205:5051 (192.168.187.205)
17:00:26.087103 Deactivating slave ...5050-5669-S0 at
 ↪  slave(1)@192.168.187.205:5051 (192.168.187.205)
17:00:26.087155 Slave ...5050-5669-S0 deactivated
17:00:37.940727 Registering slave at slave(1)@192.168.187.205:5051
 ↪  (192.168.187.205) with id ...5050-5669-S3
17:00:38.116992 Registered slave ...5050-5669-S3 at
 ↪  slave(1)@192.168.187.205:5051
17:00:38.117085 Added slave ...5050-5669-S3 (192.168.187.205)
```

Listing 1: Excerpt from `/var/log/mesos/mesos-master.INFO` showing the deactivation and the new registration of the rebooted slave. Truncated for increased readability.

Should a slave node simply be temporarily disconnected from the master node, but exceed the timeout period, the Mesos master will forcibly shut the Mesos slave node down. To account for such scenarios, the official Apache Mesos documentation recommends monitoring the Mesos slave process and restart if it should be terminated for any reason. In this case, this is achieved with a simple check using Monit. In Listing 2 log events of such a case is listed.

```
17:34:23.298998 Shutting down slave ...5050-5669-S3 due to health check
 ↪  timeout
17:34:23.300134 Removing slave ...5050-5669-S3 at
 ↪  slave(1)@192.168.187.205:5051 (192.168.187.205)
17:34:23.301009 Removed slave 20150501-230056-2407081856-5050-5669-S3
17:34:23.536837 Notifying framework ...5050-27030-0006 (marathon) at
 ↪  ...473b-b57a-83121a00a01c@128.39.121.140:43217 of lost slave
 ↪  ...5050-5669-S3 (192.168.187.205) after recovering
17:34:29.017205 Slave ...5050-5669-S3 at slave(1)@192.168.187.205:5051
 ↪  (192.168.187.205) attempted to re-register after removal; shutting it
 ↪  down
17:34:57.329751 Registering slave at slave(1)@192.168.187.205:5051
 ↪  (192.168.187 .205) with id ...5050-5669-S4
```

Listing 2: Excerpt from `/var/log/mesos/mesos-master.INFO` showing the forced shut down of the Mesos slave process at 192.168.187.205 and the registration as new slave at end. Truncated for increased readability.

The working Mesos master instance cease to function
ZooKeeper maintains an active connection to the participants of the quorum and will after a very short timeout lasting a few seconds, will initiate a new leader electing for choosing a new leading Mesos master node. As long as the number of functional Mesos master nodes is equal or higher than the quorum size, a new leader will be elected and will replace the unresponsive Mesos master node.

This scenario was tested with a simple reboot of the instance where the leading Mesos master was running. The backup Mesos masters quickly discovers the loss of connection to the leading Mesos master and promptly, with the use of ZooKeeper elects a new leading Mesos master node. The rebooted Mesos master node later joins the cluster as a backup node after coming back online.

The setup proposed in this prototype has three Mesos master nodes, with the quorum size set to two. This means that among the Mesos master nodes, one can fail without crippling the cluster, as the quorum size dictates the number of election participants that has to be able to communicate to be able to elect a new leader.

An entire region within the hybrid cloud becomes unavailable
If an entire region becomes unavailable, the Mesos nodes located within

those regions will by extension also become unavailable. In this particular case, the loss of one single site equals the loss of one Mesos master node and four slave nodes. Each node, depending on the type, is handled as specified in the test scenarios mentioned above.

This was tested by taking down the VPN tunnels at the VPN gateway of the concerned region. This cuts all communication between the the affected region and the other ones. As expected the the Mesos master nodes continued without any issues, as the current leader was not the affected one. As for the affected Mesos slave nodes, after the timeout of 75 seconds, the leading Mesos master node determined that the slave nodes were unresponsive deactivated them.

The hybrid cloud splits and semi-isolates part of the platform
In the event of split in the hybrid cloud, resulting in a partly isolated availability region, the quorum mechanics will prevent inconsistencies of the cluster and avoid issues like the split-brain problem.

To test this scenario, two simple `iptables` DROP rules was added on the Mesos master node located in Frankfurt with the IP address `192.168.0.5`. This test scenario is illustrated in Figure 5.4. The following two lines were executed at the instance:

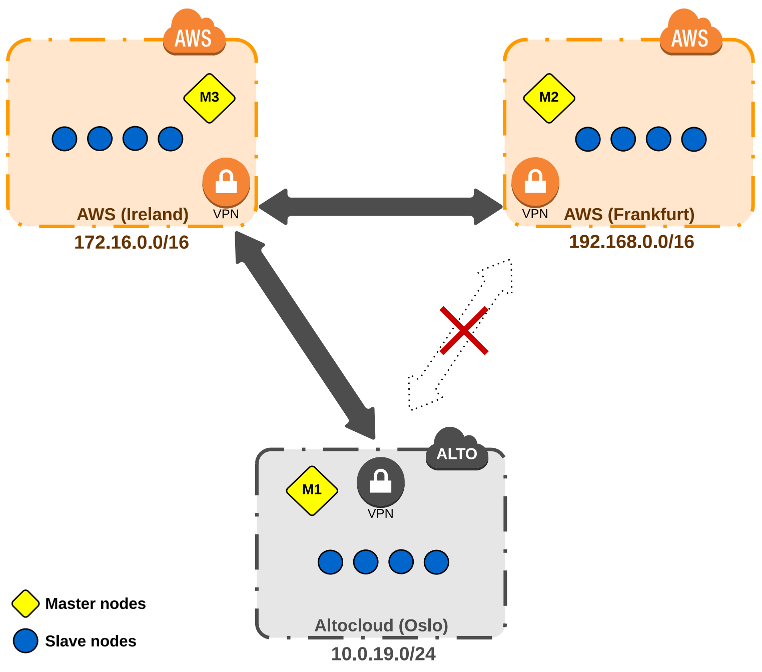
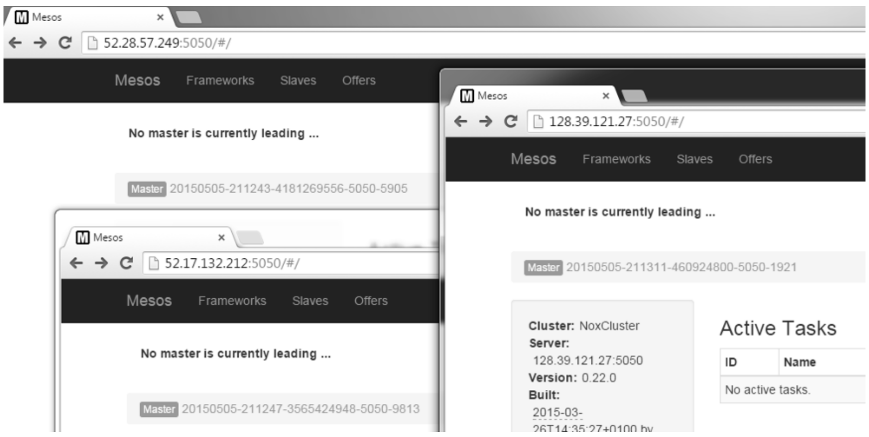


Figure 5.4: An illustration showing how the semi-isolated test scenario looks like

```
iptables -A INPUT -s 10.0.19.5 -j DROP
iptables -A OUTPUT -d 10.0.19.5 -j DROP
```


Mesos
52.28.57.249:5050/#/
Mesos Frameworks Slaves Offers

No master is currently leading ...

Master 20150505-211243-4181269556-5050-5905

Mesos
52.17.132.212:5050/#/
Mesos Frameworks Slaves Offers

No master is currently leading ...

Master 20150505-211247-3565424948-5050-9813

Mesos
128.39.121.27:5050/#/
Mesos Frameworks Slaves Offers

No master is currently leading ...

Master 20150505-211311-460924800-5050-1921

Cluster: NoxCluster
Server: 128.39.121.27:5050
Version: 0.22.0
Built: 2015-03-26T14:35:27+0100 by

Active Tasks

ID	Name
No active tasks.	

Figure 5.5: A screenshot taken over three browser windows showing each of the state of the master nodes. "No master is currently leading...".

The leading Mesos master node at the current time was `10.0.19.5`, with nothing occurring immediately as a result of the `iptables` DROP rules. The leading master continued with no issues and other two standby Mesos masters correctly redirected to the leading master node. However, after rebooting the ZooKeeper process and Mesos master process on the master nodes, the cluster is unable to elect a new leader. The cluster if effectively frozen as depicted in the screenshot shown in Figure 5.5. Immediately after the `iptables` DROP rules were removed, a new leading Mesos master were elected and operations continued as normal.

5.2.2 Prototype 2: Prioritizing local availability

Due to the architecture of Apache Mesos, it is possible to perform rolling updates on the Mesos master nodes. This ensures no downtime as each Mesos master node is updated separately and then restarted. The setup of this prototype was done by adding two more Mesos master nodes at Altocloud with the final configuration of the five master node cluster. Afterwards, each of the existing three master nodes had their configuration updated and restarted. The cluster was fully operational during this process.

As some of the test scenarios does not introduce anything new and behaves just the same as in the previous prototype, a few of those test scenarios has been omitted for this prototype.

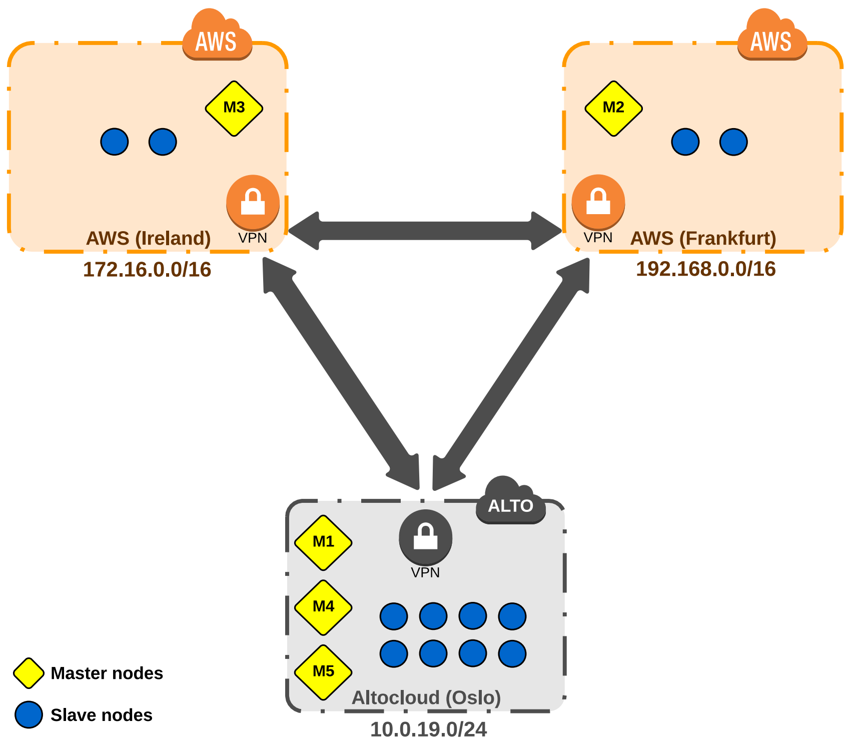


Figure 5.6: Prototype 2: Prioritizing local availability. Focusing on the availability at the local site.

Test scenarios

An entire region within the hybrid cloud becomes unavailable
The effect on cluster depends largely on which availability region that becomes unavailable. For any other region other than the private site, Altocloud, the effect on the cluster as a whole is limited. For the prototype illustrated in Figure 5.6, that would mean the loss a of single Mesos master and two slave nodes. With four Mesos master nodes left, with the majority of the slaves located in Altocloud, the cluster is fully functional with slightly less processing capability.

To verify this, the VPN gateway deactivated at Altocloud, thus simulating both the failure of the private site from the perspective of the Mesos nodes located in the public cloud and the failure of the public cloud Mesos nodes or ISP connection problems from the perspective of the private location.

The Mesos master nodes located at Frankfurt and Ireland entered a leaderless state, waiting for a leader to be elected. Since the Mesos master nodes located at the public cloud locations can not form a majority, thus satisfying the quorum size requirement, they are practically frozen while awaiting the connection of one additional Mesos Master node in order to elect a new leader.

While in the private location, Altocloud, the cluster re-elects a new leader, as the previous leader was located at Frankfurt, and continues delegating tasks and restoring those that were lost in the disconnection.

The hybrid cloud splits and semi-isolates part of the platform
In contrast to the previous prototype, a semi-split between the the regions will not cause the cluster to freeze. This is due to the majority of the Mesos master node being present at a single location. Ultimately, the Mesos master nodes present at Altocloud will sustain cluster.

However, even for this prototype, should the internal network of Altocloud be split, with semi-isolated Mesos master nodes in combination with a network split between the different regions, it is possible that the cluster will experience the same problem as prototype 1, with the Mesos master nodes being unable to elect a new leader. It is not immune to the problem although, the cluster will still run in the event of a network split occurring between the availability regions.

5.3 Segmentation of data

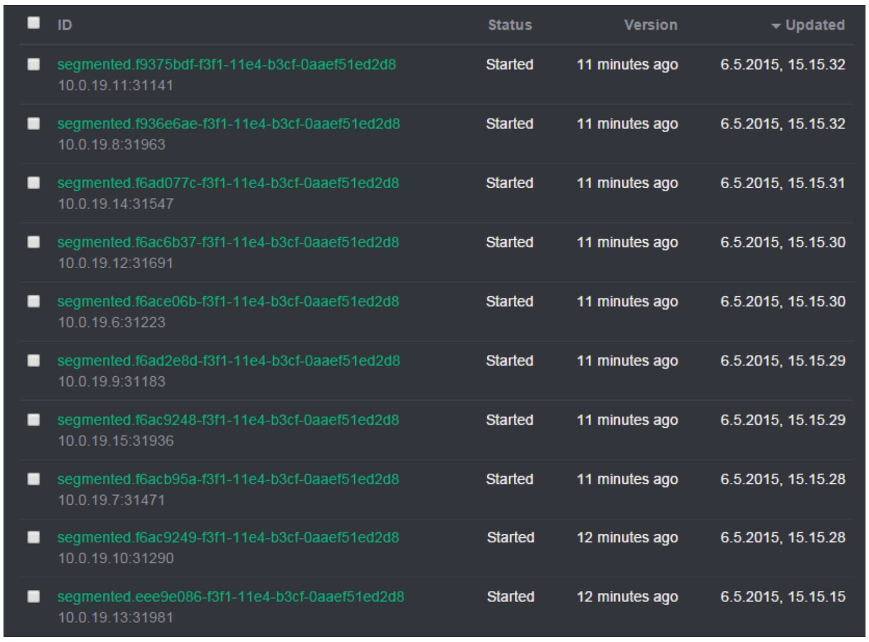
To accommodate for segmentation of specific data, Mesos slave nodes was tagged with a few attributes after installation. This was done by adding a configuration file in `/etc/mesos-slave/attributes` with the following contents:

```
cloud_type:private;country:norway;city:oslo
```

The contents of this file tags the slaves with attributes marking them with unique properties that later can be used for constraints.

To verify the segmentation of takes place the within the hybrid cloud platform, `cloud_type:CLUSTER:private` was added as a constraint for the Marathon application executed. Additionally, due to the considerable large amount of resources available, a second constraints was added, `hostname:UNIQUE`, requiring a unique hostname for each instance of the task deployed. In short, only one instance of a task can be run on a single slave nodes and the slave node has to be tagged with an attribute named `cloud_type` which is set to `private`.

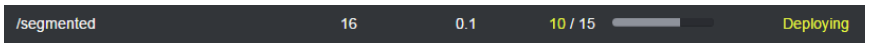
As there are 10 slave nodes located at Altocloud with the attributes set, a maximum of 10 task instances can be run at all times, unless additional Mesos slave nodes are added that fulfill the constraints. Figure 5.7 lists up the 10 Mesos slave nodes running the constrained Marathon application. Note the subnet of the IP-addresses listed, as they are part of the `10.0.19.0/24-subnet` which was allocated to Altocloud.



ID	Status	Version	Updated
segmented.f9375bdf-f3f1-11e4-b3cf-0aaef51ed2d8 10.0.19.11:31141	Started	11 minutes ago	6.5.2015, 15.15.32
segmented.f936e6ae-f3f1-11e4-b3cf-0aaef51ed2d8 10.0.19.8:31963	Started	11 minutes ago	6.5.2015, 15.15.32
segmented.f6ad077c-f3f1-11e4-b3cf-0aaef51ed2d8 10.0.19.14:31547	Started	11 minutes ago	6.5.2015, 15.15.31
segmented.f6ac6b37-f3f1-11e4-b3cf-0aaef51ed2d8 10.0.19.12:31691	Started	11 minutes ago	6.5.2015, 15.15.30
segmented.f6ace06b-f3f1-11e4-b3cf-0aaef51ed2d8 10.0.19.6:31223	Started	11 minutes ago	6.5.2015, 15.15.30
segmented.f6ad2e8d-f3f1-11e4-b3cf-0aaef51ed2d8 10.0.19.9:31183	Started	11 minutes ago	6.5.2015, 15.15.29
segmented.f6ac9248-f3f1-11e4-b3cf-0aaef51ed2d8 10.0.19.15:31936	Started	11 minutes ago	6.5.2015, 15.15.29
segmented.f6acb95a-f3f1-11e4-b3cf-0aaef51ed2d8 10.0.19.7:31471	Started	11 minutes ago	6.5.2015, 15.15.28
segmented.f6ac9249-f3f1-11e4-b3cf-0aaef51ed2d8 10.0.19.10:31290	Started	12 minutes ago	6.5.2015, 15.15.28
segmented.eee9e086-f3f1-11e4-b3cf-0aaef51ed2d8 10.0.19.13:31981	Started	12 minutes ago	6.5.2015, 15.15.15

Figure 5.7: A screenshot in the Marathon GUI listing up the running tasks at the Mesos slave nodes.

When attempting to scale further than the 10 slave nodes available, the Marathon application is unable to execute additional tasks instances, as there are no available resources that fulfill the constraint requirements. As displayed in Figure 5.8, the Marathon application is unable to scale beyond 10 tasks instances in this particular setup.



/segmented	16	0.1	10 / 15	Deploying

Figure 5.8: A screenshot in the Marathon GUI showing the Marathon application with the constraint attempting to scale beyond the available resources that fulfill the constraint requirements.

5.4 Automated cloud bursting

The main script can be found in Appendix C.

The automated cloud bursting solution uses a Python script which was developed as outlined in Results: Design chapter 4, and depends on two configuration `YAML` files and an additional small script for importing some basic functions.

The first configuration file is the main configuration file and contains the main preferences used by the script. Amazon Web Services API credentials, execution interval, and maximum limits are just a few of those settings. The second configuration file is the launch configuration file and contains the properties that is used when launching a spot instance.

The particular setup of the hybrid cloud platform followed the prototype 2, as outlined in Figure 5.6, with the exception of the Mesos slave nodes located at the public cloud providers. The Mesos slave nodes located in Ireland and Frankfurt were deactivated prior to the experiments. The setup utilizes 10 Mesos slave nodes as the baseline resources located in the private location, Altocloud. Each of the ten slave nodes were running on a m1.large instance type, resulting in 40 CPUs, 80 GB of memory, and 800 GB of storage in total.

The script is designed to be executed as Marathon application and depends on being present in the cluster, as it attempts to discover to leading Mesos master node dynamically each iteration. Except for this, the script can be executed as any other python script from the bash prompt and accepts the following arguments:

```
usage: burst.py [arguments]

Arguments:
   --help                       Prints this help message
   -v [--verbose]               Verbose output
   -c [--config]                Specify another config file
```

The automated cloud bursting solution consist of three main parts: data collection, price and scaling decision logic and management of the spot requests and spot instances.

Before any data is collected, the script load the configuration file into the script and use the preferences retrieved to resolve a ZooKeeper URL to find the current leading Mesos master. The script then proceeds to collect resources metrics from the leading Mesos master before moving on to collecting information about any active spot requests and instances from Amazon Web Services EC2.

If the verbose flag was set when executing the script, the following information about resource usage is displayed in the terminal:

```
|-------------------------------------------
| Resource usage: |  Percent      | Count
|-------------------------------------------
| CPUs            |  2.68%        | 1.10
| Memory          |  0.25%        | 176 MB
| Disk            |  0.00%        | 0 MB
|-------------------------------------------
```

As previously stated, the bidding strategy chosen was the *current price + x* and was implemented by polling the price each iteration through the EC2 API. In addition, a maximum bid limit, specified in the configuration file, is checked before returning a bid value. As the prices fluctuate, so does the bid the script up until the maximum limit.

The decision to scale or not it determined on the current resources usage percent compared to the burst point threshold. The script ensure that that the number of spot instances that serves as a Mesos slave node corresponds to the numbers required to keep the usage percent just below the burst point threshold. As with the maximum bid limit, there is also a maximum spot slave limit which limits the number of spot slaves the script is allowed to scale up to. With the value set to 0.75, the script will automatically burst into the cloud and start requesting spot instances if the the limit of 75% of any single resource is exceeded. The script will evaluate the usage percentage with the current active Mesos slave nodes in addition to the pending resources to calculate the percentage. To scale down, the script will calculate the usage percentage of the cluster after removing x amount of slaves before in order to find the optimal number of spot instance Mesos slaves with respect to the burst point threshold.

The script will also cancel older spot instance requests that exceeds a certain time limit, which is set in the configuration. This is done in order account for spot instance requests that for some reason is stuck and will not result in any spot instance any time soon. This could be due to an error or that the price bid is too low. In the use cases of a cloud bursting solution, the requested resources should appear within a reasonable amount of time, otherwise it would not serve the purpose as a cloud bursting solution.

The activity diagram shown in Figure 5.9 describes the logic which determines the scaling action.

5.4.1 Scaling in action

To showcase the functionality of the cloud bursting solution, two experiments has been conducted. Each experiment covered a unique use case which is interesting in a cloud bursting scenario.

To simulate heavy workloads, several Marathon applications were created with the sole purpose of hogging available resources. Below is an example of the commands used for resource-hogging tasks.

```
while true; do echo hello world; sleep 60; done
```

As evident by looking at the command, it does not by itself consume a lot of resources to run. However, each tasks are allocated a certain amount of resources by the Marathon framework and can within those allocated resources use as little or as much as needed. For the simulations each of these small `while-loop` tasks are given excessive amounts of resources in order to quickly hog up large quantities of resources for the simulation.

There are three possible resources variables to simulate:

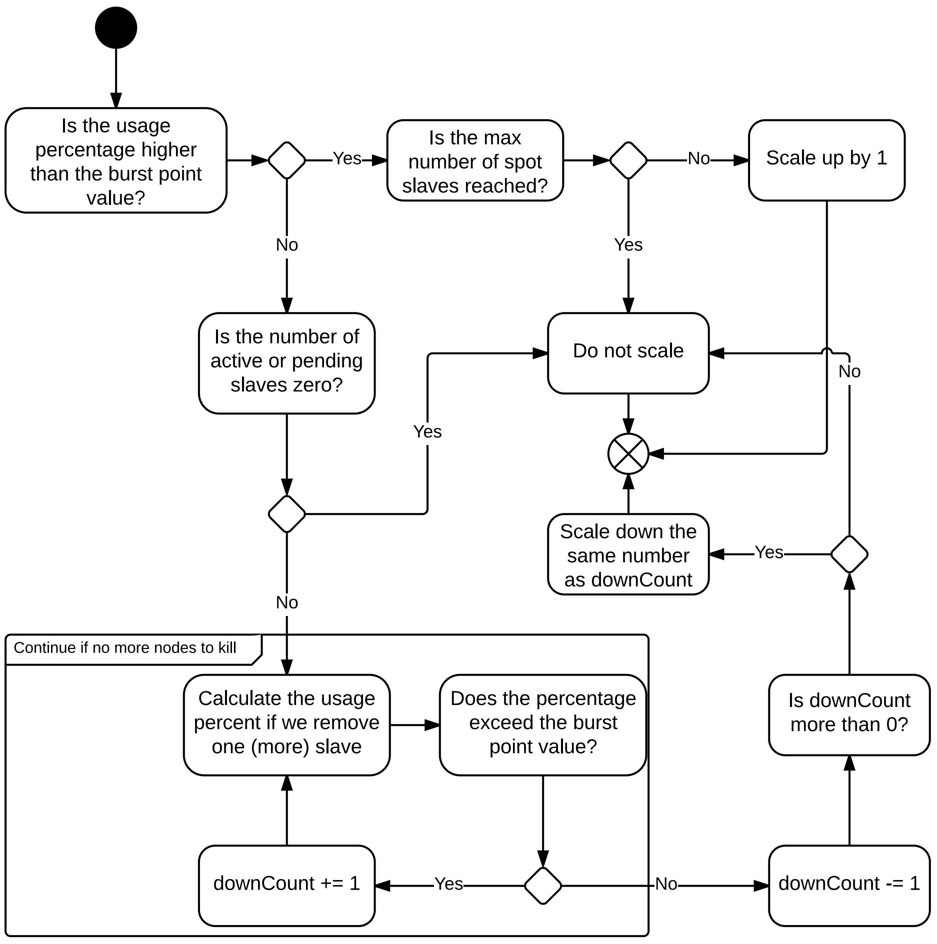

Is the usage percentage higher than the burst point value?
Yes
Is the max number of spot slaves reached?
No
Scale up by 1
No
Yes
Is the number of active or pending slaves zero?
Do not scale
No
Yes
Scale down the same number as downCount
Yes
No
Continue if no more nodes to kill
Calculate the usage percent if we remove one (more) slave
Does the percentage exceed the burst point value?
Is downCount more than 0?
downCount += 1
Yes
No
downCount -= 1

Figure 5.9: An activity diagram showing the decision logic for determining whether or not to scale.

- CPUs
- Memory
- Disk

CPUs as a resource is normally denoted with integer numbers and was therefore picked as the resource to exhaust in order to test the cloud bursting solution. To do so, a Marathon application named *hog-cpu-1* was created and scaled up to 100 tasks, effectively hogging 100 CPUs. The baseline resources consist of 40 CPUs and any additional resource required will there depend on resources available through the cloud bursting solution. With the maximum amount of spot slaves set to 10, the cluster can not accommodate for 100 CPUs while scaling up on the lower tier instance types, resulting in a constant need for resources.

Experiment 1 - Scaling up

Table 5.4, contains the parameters which was used when conducting the experiment 1. The goal of this experiment is the showcase the cloud bursting functionality, more specifically the scale-up part of the script. At the beginning of the experiment, `hog-cpu-1` will be scaled up to 100 tasks in the attempt have the script scale up to 10 spot instances to serve as Mesos slave nodes.

Variable	Value
Execution interval	60 seconds
Instance type	m3.medium
Maximum spot slaves	10
Maximum bid limit	0.500
Burst point percentage	85%
Spot request timeout	10 minutes
Price padding	0.001

Table 5.4: The parameters for the cloud bursting experiment 1.

As seen in Figure 5.10, the cloud bursting script scales up the desired number of slave nodes to 10, alternating between 10 and 9. During runtime, the script calculates a bidding price each iteration, which is the current market price along with a padding of 0.001. However, the number of instances registered to Apache Mesos alternates between one and zero. This is due to the increasing market price, which invalidates the previous spot instance requests, as the bidding price for those instances were lower than the current market price. As a result, the instances are terminated shortly after they are booted up.

After 30 minutes, the CPU-hogging tasks were halted, thus resulting in almost no resource requirements for the cluster. As a result, the number of desired slave nodes as well as the bidding price is 0, as no slave nodes are needed. A few minutes later the current market price falls to the original prices before the scaling experiment took place.

Experiment 1 revised- Scaling up

As the original experiment 1 did not showcase the scripts ability to scale up the instances, a number of tweaks to the configuration were made for a revised experiment. As the price changes deterred the script from properly bursting into the cloud within 30 minutes of time, a revised experiment was deemed necessary. Table 5.5 lists up the tweaked parameters for the revised experiment.

Due to the very volatile market price for the `m3.medium` instances type in Frankfurt, `c4.large` instance type was chosen instead. Compared to `m3.medium`, the market price for `c4.large` instances was far more stable. Additionally, as the goal is to verify the scaling functionality of the script,

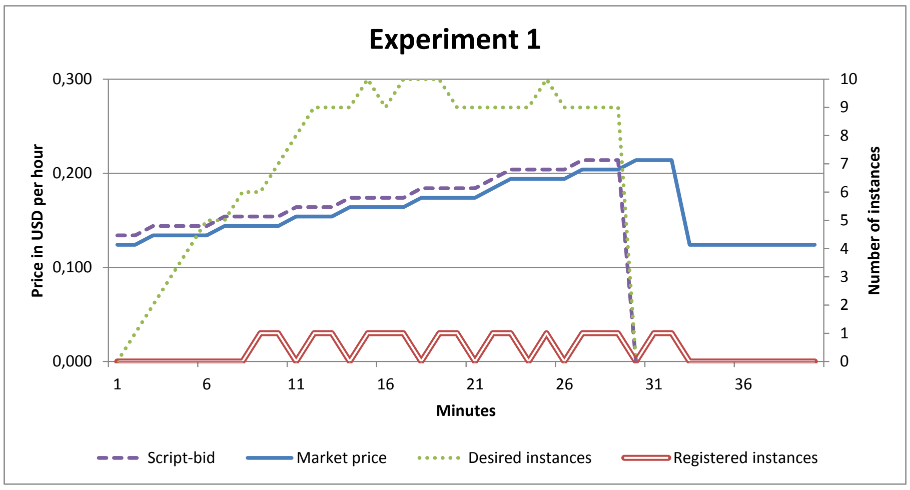


Figure 5.10: Experiment 1: The market price rises at the same interval as the cloud bursting script does and drops as soon as the script stops bidding.

the price padding was set to 0.100 as an aggressive measure to deal with any spikes to the market price.

Variable	Value
Execution interval	60 seconds
Instance type	c4.large
Maximum spot slaves	10
Maximum bid limit	0.500
Burst point percentage	75%
Spot request timeout	10 minutes
Price padding	0.100

Table 5.5: The parameters for the revised cloud bursting experiment 1 and experiment 2.

As seen in Figure 5.11, the script managed to successfully request spot instances, with the Mesos slave nodes registering in bulks. After approximately 15 minutes, all the requested slave nodes were online and processing for the cluster. The price remained the same for the entire duration.

Experiment 2 - Scaling down

The goal of of this experiment is to showcase the script in a scale-down scenario. The configuration used for experiment 2 was the same the revised experiment 1 had, listed in Table 5.5. This was to ensure the cloud

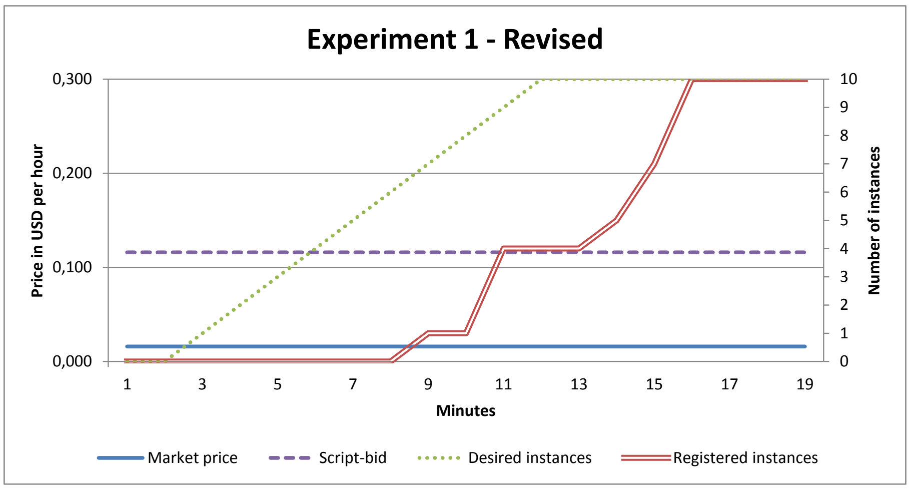


Figure 5.11: Experiment 1 revised: A successful scale-up experiment with the market price being stable for the entire duration of the experiment.

bursting script to successfully scale down without any major disruptions. Additionally, Table 5.6 contains specific parameters relevant for experiment 2 which.

In this experiment, the spot instances that were active in the revised experiment 1 where used to scale down.

Variable	**Value**
Partial hour threshold	20 minutes

Table 5.6: Additional parameters in the configurations set for experiment 2.

Figure 5.12 shows the cloud bursting script downscaling the number of spot instances from 10 down to 0. The number of `hog-cpu-1` tasks were scaled from 59 tasks down to 24 in the interval of 5, with 24 CPUs being just below the limit before the script will burst into the cloud. As seen in the graph, although the desired number of spot instances is zero, the script keeps them alive for a longer period of time before scaling down. This is due to the imposed requirement that requires each instance to be alive for at least x amounts of minutes in an hour-cycle, where x is the partial hour limit set in the configuration. For this experiment, the limit was set to 20 minutes in order to make the waiting period for this experiment shorter.

Listing 3 is an example of an single iteration of the script. In this case there are five active spot instances serving the cluster as slave nodes. However, none are desired as the current resource usage is below one

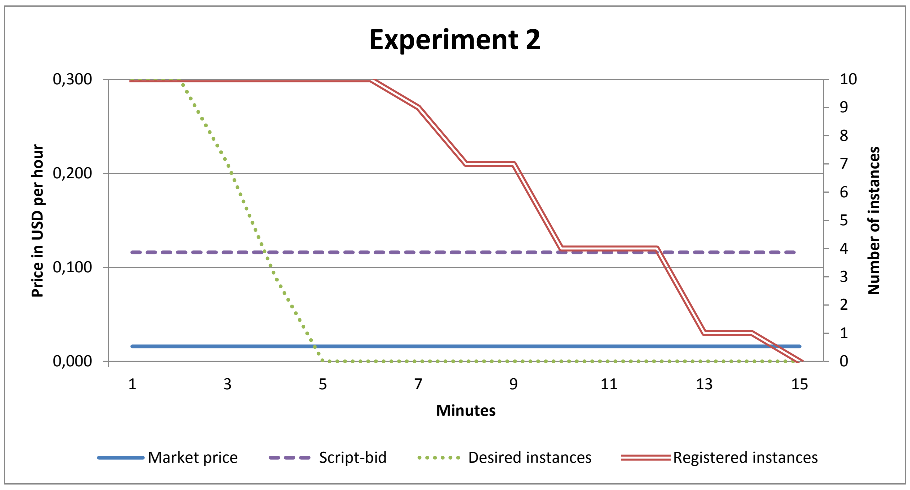


Figure 5.12: Experiment 2: A successful scale-down experiment with the script waiting until the specified minimum time spent in an hour-cycle before terminating the instances.

percent for all three categories of resources. In this case, even the baseline resources available at the private location is mostly idle. The script attempts to terminate the excessive slave nodes however, since the number of minutes since last integer hour has not passed the specified limit, the script postpones the termination of the instances. The limit for the least amount of minutes that is required to pass before terminating an instance is set in the configuration file.

```
Resolving ZooKeeper url for the working Mesos Master
   Current Mesos master 128.39.121.27
Fetching http://128.39.121.27:5050/metrics/snapshot
   |--------------------------------
   | Curent market price:    0.026
   | Maximum bid limit       0.050
   | Our bid                 0.027
   |--------------------------------
   |------------------------------------------
   |      Burst point value set to 0.85
   |------------------------------------------
   |------------------------------------------
   | Resource usage: |  Percent         | Count
   |------------------------------------------
   | CPUs            |  0.22%         | 0.10
   | Memory          |  0.02%         | 16 MB
   | Disk            |  0.00%         | 0 MB
   |------------------------------------------
   |----------------------------
   | Number of:         | Count
   |----------------------------
   | Desired instances  |   0
   | Pending requests   |   0
   | Active instances   |   5
   |----------------------------
Excessive spot instances. Attempting to terminate 5
i-4bf24b8a has not reached the set partial hour limit. 16 minutes has
 ↪  passed.
i-6ff24bae has not reached the set partial hour limit. 9 minutes has passed.
i-84f54c45 has not reached the set partial hour limit. 9 minutes has passed.
i-bff54c7e has not reached the set partial hour limit. 9 minutes has passed.
i-16f24bd7 has not reached the set partial hour limit. 3 minutes has passed.
The script used 0.615634202957 seconds this loop
Sleeping additional 59.384365797 seconds
```

Listing 3: The verbose output of one iteration in the cloud bursting script. Five instances are excessive and attempted terminated, but is ultimately postponed due to the lifetime of the instance.

Chapter 6

Analysis

This chapter contains the analysis of the results written in chapter 4 and chapter 5 and covers the analysis of the proposed designs, the implemented solutions, and the test and experiment results.

6.1 The testbed and VPN

The testbed consist of Altocloud as the private site and Amazon Web Services VPC regions as separate external sites. An interesting consideration regarding this is how close to the hardware one should install Apache Mesos. In this project, the cluster was installed on an OpenStack installation, thus on top of a virtual environment. This entails some extra overhead that may not result in any distinct advantages for Apache Mesos directly over a bare-metal installation. In this thesis, focus on performance is not within the scope and the use of a virtual environment makes it easier to request and manage resources, as it is shared among other users at the University College. However, in a production environment, a bare-metal installation should be considered as performance is a vital part of any data center regardless of the form factor.

However, there are cases where the use of virtual machines is needed and preferred. For instance, when migrating over to a cluster solution like Apache Mesos, it would make sense to run Apache Mesos on virtual machines, as the same resources can be used to accommodate other services that does not run on top of Apache Mesos. This is particularly true for services running on top of non-Linux operating systems like Microsoft Windows.

Though in the end, the most obvious use case for an Apache Mesos cluster is when data center-like properties are required. Easier management, large amounts of pooled resources, as little overhead as possible and a lean interface to interact with for developers and system operators. In this use case, Apache Mesos would benefit from being installed on bare-metal, as it results in less complexity and overhead for the system operators to deal with.

The testbed was mostly installed and configured manually however, in a production environment the use of a configuration management would be more preferred. Configuration management systems will streamline the installation process and ensure the environment is updated and in the desired state.

6.1.1 VPN

Due to the way the OpenStack installation, Altocloud handles floating IPs and Amazon Web Services handles elastic IPs, coupled with the way Apache Mesos communicates, several VPN tunnels had to be set up as mentioned in Results: Implementation chapter 5.

VPN emulates a private network and solves not only the issue of the cluster, but also adequately secures the transmission, thus making TLS encryption support not as vital. However, the network is most likely also used by other services and a malicious user could potentially sniff the packets to uncover the contents of the traffic. A malicious user could potentially perform a man-in-the-middle attack to manipulate the contents of the tasks sent to set up malicious tasks in the Apache Mesos cluster. TLS encryption would add an another layer of encryption to Apache Mesos traffic and it would be more difficult to read and manipulate the traffic between the Mesos nodes.

Apache Mesos is arguably designed and intended to be used in a single location with low latency networking, which is why the use of NAT-like functionality like floating and elastic IPs does not work well with Apache Mesos. Additionally, for traffic that is transmitted over the Internet, setting up VPN tunnels is considered good practice, as it encrypts all the traffic between the VPN sites.

However, setting up VPN potentially introduces a single point of failure to the whole cluster, depending on the way the cluster is set up. To mitigate this, the VPN tunnels should be set up in a redundant manner with multiple VPN connections between the sites, possibly with with separate *Internet Service Providers* (ISP) for each link.

Additionally, to set up a site-wide VPN connection, some degree of network manipulation has to be possible. Due to this, a VPN solution may potentially exclude some public cloud providers that does not support internal networking. Although it is possible to route traffic through the Internet to an instance at the same cloud provider functioning as a VPN gateway, it defeats one of the main purposes of a VPN, namely secure transfer.

6.2 Availability

As stated in the Approach chapter 3, theoretical simulations and calculations will be done analyze the availability of the hybrid cloud platform. Table 6.1 lists up the failure rates used in the simulations to verify any increased availability due to the hybrid setup. The failure rates are artificially set and does not represent the actual failure rates of any services.

Locations	Failure rate in a year
Site A: Altocloud	15%
Site B: AWS Ireland	2%
Site C: AWS Frankfurt	5%

Table 6.1: Artificially set failure rates for Altocloud and a two Amazon Web Services availability regions.

Due to a mathematical property regarding probability. If $P(x)$ is independent of $P(y)$, then the probability of both x and y occurring is $P(x) \cdot P(y)$. Which, will always be equal to or lower than $min(P(x), P(y))$. This is called the *multiplication rule*.

As long as it can be established that the chance for the locations to fail are independent of each other, then the probability of each of them failing at the same time will always be lower due to the multiplication rule. With redundant power lines and ISP links to the Internet, the hybrid cloud platform can achieve high availability from the very bottom of the stack with hardware, all the way up to each application.

However, there are other sources of failures that are not related to the sites directly. At the same time as independence between the locations are preferred, several other vectors for failure are introduced, with the Internet being a particular possibility. Apache Mesos makes the platform fault-tolerant for failures at the hardware level, while distributing the Mesos master nodes between independent sites makes the platform fault-tolerant to site-wide failures. However, it does introduce the additional risk of losing network connectivity.

There are also extreme cases like natural disasters or a nation wide or larger outage of electricity or damage to an intercontinental backbone connection. In those cases there is little a single organization can do to mitigate and service disruptions of this level would most likely affect the end users as well. Fortunately, the chances for extreme cases like this are low, at least compared the probabilities for a single cloud location becoming unresponsive.

Consider the following: A single location data center with a cluster running is only dependant the internal networking and the hardware within the data center. The moment an external dependency is introduced, the Internet becomes a factor. Depending on the use case of the cluster, that may result in additional risks being added, thus lowering the expected availability. For a cluster that is meant to be accessed from the Internet, with public facing services like websites, this does not introduce additional risk, as Internet is already a part of the equation for these type of use cases.

For the purpose of the simulations, several assumption has been made:

- Each location is 100% independent of each other, meaning that the probability of failure in one location does not affect the probability of any of the other locations.
- There are no other causes for failures on the hybrid cloud platform

6.2.1 Prototype 1: Maximizing availability

The setup outlined in prototype 1 was designed for maximizing availability for a three Mesos master node setup. However, this setup mainly accounts for hardware issues and problems with the Mesos processes and does not include the aspect of networking issues to the same degree. On the other hand, the cluster remains accessible as long as at least two of the sites are responsive which, unless there are major issues with backbone-tier Internet infrastructure is not very likely to occur, as the location of the sites are geographically spread.

While the this particular cluster setup can survive a national-level of networking failure, it is important to consider the target audience for the the hybrid cloud platform. If the end-users are mostly contained within a single country, the sites should, if possible be located within the country or at least close neighboring countries, to improve latency and make the hybrid cloud platform independent on other countries infrastructure, as it is irrelevant for the target audience in this case. The setup described in prototype 1 would work well for an European target audience, as the sites are spread across the continent of Europa.

The probability for the entire hybrid cloud platform to become unavailable is when at least two of the sites becomes available. Assuming the probability for failure is as listed in Table 6.1, the highest probability for the cluster to fail is the combination of the private location, Altocloud and the public cloud location, Amazon Web Services Frankfurt to fail simultaneously. The calculations therefore becomes:

$$P(A_{failure}) \cdot P(C_{failure})$$
$$0.15 \cdot 0.05 = 0.0075$$

Even with the two sites with the highest likelihood for a failure, the probability of them occurring at the same time is lower than 1%. The probability for failure with any other combinations of the sites that results in the hybrid cloud platform do become unavailable are even lower.

A Mesos slave process becomes unavailable

Depending on the workload deployed in the cluster, one or several Mesos slave nodes can become fail without affecting the cluster as whole. Though, the tasks that were running on the affected Mesos slave nodes will be lost and rescheduled for other Mesos slave nodes with available capacity. The main causes for one or few slave nodes to fail would most likely be due to a hardware issues, as the timeout of 75 seconds rules out the possibility of smaller and temporary network issues. It could also be due to a freeze or a crash in the Mesos slave process.

The working Mesos master instance cease to function

In this prototype, the number of Mesos master nodes that can become unresponsive while still maintaining the cluster is one. To elect a leader, ZooKeeper requires a majority vote to be conducted with the the excepted number of ZooKeeper participants as baseline. A new leader is elected as long as there are two ZooKeeper participants left.

As for the Mesos slave nodes, they have to re-register at the new master within a set time frame. The default value is set to 10 minutes. Any slave nodes that fails to do so is marked as unresponsive and is deactivated. Even in the event of a leaderless cluster, the slave nodes will continue to process whatever tasks that were given and a separate timeout period is tracked at the Mesos slave node, which is by default set to 15 minutes. If the slave node does not manage to re-register with a leading Mesos master node within this timeout, the Mesos slave process will self-terminate.

An entire region within the hybrid cloud becomes unavailable

For the setup outlined in this prototype, the effects of an entire region becoming unresponsive will be limited, as each region is equal in both the number of Mesos master nodes and slave nodes. As long as the remaining slave nodes can handle the workload of the cluster by themselves, the cluster survives and continues to process tasks. Even if the remaining cluster should lack the capacity to fulfill the entire workload, the tasks will be queued up and prioritized depending on how the framework running on top of the cluster has been designed. Apache Mesos will prioritize between the frameworks depending on prespecified prioritizing, should multiple frameworks be present.

The hybrid cloud splits and semi-isolates part of the platform

A semi-isolated network can happen for various reasons. For instance it could be due to issues with backbone-tier connections that result in a highly throttled connection that could prioritize certain traffic, thus semi-isolating certain networks. However, the Internet is designed to be quite resilient in terms of connection faults and a packet should eventually be routed to its destination as long as a path exist. With a VPN solution a split network is possible if the the specific VPN tunnel between two sites goes down for some reason. It could also be due to security systems wrongfully flagging traffic from a certain source IP-address as malicious and therefore dropping it.

The testing the case of a semi-isolated network produced some unexpected results. The Mesos masters were unable to elect a new master in the semi-isolated network scenario with the Mesos master node located in Ireland unable to reach consensus with two isolated networks communicating to it. As the Apache Mesos cluster was functioning until a new election had to be made, it would seem that the issue originates from ZooKeeper alone.

In order to mitigate this, additional Mesos master nodes can be added, thus forming a five, seven or larger Mesos master node cluster. A single semi-isolated network is not enough to affect the cluster, as the number of participants for the ZooKeeper leader election would exceed the quorum size requirement. While it is possible to experience the same issue with clusters with more than three Mesos master nodes, it would arguably be less likely to happen, as it requires the network to partition itself in such a way that leaves a single Mesos master node dealing with equal parts of isolated networks.

6.2.2 Prototype 2: Prioritizing local availability

As stated in Results: Design chapter 4, this type of setup sacrifices overall availability for the purpose of prioritizing local access. With this setup the majority of the Mesos masters are located in the private location, thus making that particular site a single-point of failure. The cluster has a failure rate equal to the private location where the majority of the Mesos masters are located which, with the failures rates set in Table 6.1, would mean 15%.

On the other hand, this setup would guarantee that the cluster is running as long as the private site is functioning. This makes this type of setup ideal for cloud bursting scenarios where a baseline resources are running services with the added option to supply the cluster with extra resources and heightened availability, without depending on the the external resources.

In the event of issues with the ISP for the Altocloud location, the cluster will isolate itself and keep the cluster available for those with access to the local network. The external parts of the cluster will not be able to form a cluster, due to the required quorum size.

Whereas, if Altocloud becomes unavailable, the entire cluster will be suspended, as the majority of the Mesos master nodes are unresponsive.

Additionally, since most of the processing is done at Altocloud by the 10 slave nodes located there, a considerable amount tasks becomes lost.

6.3 Segmentation of data

As written in the Approach chapter 3, there is an assumption that the internal Mesos traffic follows a certain pattern regarding the flow of information outlined in Figure 3.2.

The use of constraints adequately segments tasks as shown in the results. For use cases where the purpose of segmentation is not related to privacy concerns, it suffices as a solution. However, if privacy is the priority, then several challenges arises.

In terms of use cases where privacy is the priority, there a weakness of the solution that needs to addressed. With the framework Marathon, the default constraint is no constraint, thus the default behaviour is to spread the workload evenly. This may pose a problem for use cases where tasks are sensitive in nature. This could for incautious developers or system administrators result in private data leaking out and thus breaking regulations or privacy interests.

As an alternative, Apache Mesos can be deployed at compliant sites. his would mean private locations and certified cloud providers for the purpose. This way, data segmentation becomes irrelevant for the cluster. Amazon Web Services has an availability region named *GovCloud* which satisfies several U.S. requirements for government agencies to be able to use. Other cloud providers in other nations may have the same type of product that meets requirements for other use cases or nations.

If a cluster with the possibility to leverage public cloud provider is still desired, a separate cluster can be maintained as an alternative. One being a hybrid for cloud bursting and public services and one for sensitive workloads that needs to be separated for compliance or privacy needs. However, this would mean additional maintenance to be done and may not work for the use case or be too expensive.

In the event that the regulations or privacy requirements only concerns data storage location, and not processing of the data, the use of a hybrid cloud may be an viable option, with or without constraints.

6.4 Automated cloud bursting

The automated cloud bursting solution was written as prototype to showcase the possibilities with automated cloud bursting coupled with spot price instances. The main focus of the script was to adequately design, prototype, and test a solution adequate for answering the problem statement.

The script is built on the assumption of that cloud bursting is is not critical for the operation of the services and merely an addition which is preferred to have. One of the reasons for this is the delays of several minutes incurred when requesting spot price instances, as the the requests has to be processed and then a machine has to boot up and register itself to the cluster. This may take from 5 to 10 minutes, depending on Amazon Web Services, as spot requests seems to be processed in batches every few minutes or so. For a more prompt solution, on-demand instances should be considered, as they are booted up almost instantly.

Furthermore, to bootstrap the the slave nodes, a simple bash script is supplied as user data at instance launch. The script will install and configure the necessary services to make it register itself to the cluster. This adds an approximate of three to four minutes of overhead when deploying a Mesos slave node. This overhead would most likely not be improved with the use of any configuration management tool, on the contrary, it would most likely add additional overhead. In order to shorten this time, an image can be prepared beforehand with the necessary services configured.

During the experiments, a default execution interval of 60 seconds was used. The reason for such a lengthy execution interval is due to the slow update rate regarding spot requests at Amazon Web Services EC2. It takes around 20-30 seconds at most for a spot request to become registered and in order to prevent duplicate requests, the execution time has been set to 60 seconds. In addition, the length of the execution time allows the script to follow price fluctuations better, as the time granularity is higher.

6.4.1 Experiment 1

The first attempt of experiment 1 ended up in a failure in terms of scaling up the desired number of Mesos slave nodes within a reasonable amount of time. Due to the script bidding at very minimum of the current price + 0.001, the instances were constantly terminated as the prices rose. Although the experiment was a failure in terms of scaling up within a reasonable amount of time, given an infinite max bid limit, the script would eventually stabilized at a high enough bid to maintain a spot instance for more than a few minutes. Interestingly though, the current market price during the experiment increased with 0.010 approximately every three to five minutes in steps for the entire duration of the experiment. Shortly after terminating the `hog-cpu-1` tasks, the price fell down to the price before the experiment started as seen in Figure 5.10.

Note that the market price for `m3.medium` during the experiment was well beyond the on-demand price of 0.083 USD per hour for that particular availability zone. By looking at the the price history for that particular type of spot instance, the more common price level is approximately 0.012 USD, which makes the market price during the experiment about nine times as high than normal.

It is possible to make spot instance requests persistent, meaning that a request will persist even if a spot instance is terminated, effectively re-requesting spot instances should they be terminated. This feature possibly combined with a lot of competition for the resources may caused the rise. Another possible explanation for this kind of behaviour could be that other users of spot instances also have some kind of dynamic price bidding system in place, dynamically bidding on the spot instances.

Interestingly, due to the characteristic of Amazon Web Services not charging for any partial hour used, nothing was billed for the spot instances during the experiment, even if some amount of processing was done for the Apache Mesos cluster.

In order to successfully showcase the functionality of the script, a revised experiment was conducted. As the instance type `m3.medium` had a very volatile price fluctuations for the current market price, `c4.large` was picked in instead. `c4.large` had a rock stable market price of 0.016 for the entire day the experiment was conducted and was therefore picked. In addition a lot more aggressive bidding price of 0.100 higher than the current market price was used. In the end, the aggressive price bid was not necessary, as the market price remained stable for the entire duration of the experiment. In addition, in order to get a more stable behaviour in terms of the desired amount of slave nodes, the burst point threshold was adjusted to 0.75, resulting in the cloud bursting solution to scale up with lower workloads on the baseline resources.

In contrast to the first attempt of experiment 1, the script managed to successfully scale up the number of Mesos slave nodes to the desired number after approximately 15 minutes. Spot requests are processed by Amazon Web Services in batches, resulting in the Mesos slave nodes registering in batches as well, as evident in Figure 5.11.

6.4.2 Experiment 2

Experiment 2 covers the the scenario of scaling down the number of Mesos slave nodes serving from a spot instance. When the resource usage drops below the burst point threshold, the script attempts to terminate excessive instances. However, as seen in Figure 5.12, the script does not immediately terminate the instances, but waits until the partial hour threshold has passed before terminating. This ensures that cluster will utilize the resources for the hour charged, as the instances that are terminated by user-initialization will be billed by the hour rounded up.

For this experiment, the threshold was set to 20 minutes however, in a production environment the threshold should be set to a value that is just below 60 minutes in order to fully capitalize on the hour charged. As

Amazon Web Service EC2 instances are not truly terminated before the instance is fully shut down, a few minutes should be incorporated into the threshold.

Chapter 7

Discussion

This chapter contains discussion about the results with respect to the problem statement, the process of the project, additional considerations, and impact of the findings.

7.1 The problem statement

First and foremost: the main goal of the thesis was to design and implement a hybrid cloud solution as described in the Approach chapter 3.

For convenience, the problem statement is listed below:

How can we build highly available, segmented computer clusters using private computer hardware together with public cloud providers as a hybrid cloud platform, leveraging spot price instances for an automated cloud bursting solution?

Proposed solutions for the different aspects of problem statement have been made, each attempting to address the challenges of availability, segmentation and cloud bursting for a hybrid cloud environment. By abstracting the resources located in heterogeneous cloud environments, using Apache Mesos, practical solutions has been prototyped that collectively addresses the problem statement.

By combining data segmentation and automated cloud bursting with the proposed prototypes on availability setups, two possible combinations are possible. Prototype 1 for maximizing availability and prototype 2 for prioritizing local availability. Both setups are viable and addresses the problem statement adequately.

The sub-questions defined to answer the problem statement in the Approach chapter 3 is each answered below.

7.1.1 Hybrid cloud

How can one build a computer cluster on top of private computer resources in addition to multiple public cloud providers?

In this thesis, Apache Mesos was used as an abstraction layer between the resources and the higher layers where the applications reside to tie together heterogeneous cloud locations.. Since Apache Mesos does not support NAT network configurations, VPN was installed. With VPN abstracting the network layer for Apache Mesos a computer cluster was successfully deployed in a hybrid cloud configuration using private computer resources and public cloud resources.

As mentioned in Results: Design chapter 4, without VPN, the proposed prototypes for setting up the hybrid cloud relies on a vast amount of Internet routable IP-addresses. With the IPv4 address space being full, it is difficult to acquire a large amount of IPv4 IP-addresses that would be required for a larger hybrid cloud platform. These problems may be solved in the future as a result of IPv6 support in OpenStack, Amazon Web Services, and Apache Mesos. With IPv6 IP-addresses, every node in the cluster can have their own unique Internet routable IP-address. For securing the communication between the nodes, IPSEC or another encryption method can be used. As mentioned in the Background chapter 2, TLS support for Apache Mesos is upcoming and may be adequate for securing the traffic.

However, depending on the use case, VPN may still be the preferred option, as it isolates and simplifies the network. Though, it may affect the performance and should be investigated if this is of great importance.

7.1.2 High availability

How can one, by utilizing both a private hardware and public cloud providers, gain improved levels of availability?

This thesis outlines some possible ways to set up a hybrid cloud using Apache Mesos to achieve higher availability than on a single cloud provider or data center location. By deploying a hybrid cloud on multiple and independent locations, the risks for the entire cluster to fail is distributed and lowered as result of the characteristics of independent probabilities. However, by relying on distributed locations, network becomes a factor in the equation.

This is the reason for why two prototypes was designed and evaluated. One prototype that maximizes availability in general with no concern for where the data is accessed, and an another prototype that prioritizes local access.

With prototype 1, Mesos master nodes are distributed between independent sites for reducing the failure rates of the Mesos master nodes in specific. By modifying this prototype slightly by adding additional independent sites and additional Mesos master nodes, the failure rates are in theory decreased for each node added. However, for each independent site, another dependency is added in the form of dependency of network connectivity. Though, the failure rates for the network connectivity through the Internet would mainly be regarded as an independent probability, separate of the failure rates of the Mesos nodes location. This could therefore reduce the probability for failure for the hybrid cloud platform as a whole if the number of sites are sufficient.

Prototype 2 on the other hand, prioritizes local availability. Here the focus is on the local access that is independent of external influence. Should Internet access be lost, this setup will allow the cluster to be run and accessed from the network of the private location. This setup considers the external cloud providers as an addition, and not as a critical part of the cluster. This type of setup works well with cloud bursting use cases, as the private data center functions as a preferred baseline, with the option to utilize public cloud resources for additional processing power.

Additionally, the discovered issues regarding semi-isolated networks for prototype 1 in particular, the availability improvements may not be as high as initially expected. Due to the issue with semi-isolated networks, a three master node cluster may not be sufficient, and additional master nodes may be required in order to partly mitigate this issue. While semi-isolated connectivity problems is not all common on the network in general, the use of VPN tunnels makes it a more likely event to occur, as separate VPN tunnels between the sites may in theory fail independently of each other. Consequently, any final setup should account for a semi-isolated network and the issues that arises, thus a three Mesos master node setup may not be good enough for hybrid cloud usage. Five or seven Mesos master nodes should be considered where a semi-isolated network is prone to happen.

7.1.3 Data segmentation

How is it possible to segment data and data processing to specified locations or groups?

Apache Mesos accommodates for segmenting data into parts of the hybrid cloud platform using attributes for tagging the Mesos slave nodes and the constraints feature of the Marathon framework. This has also been verified, as no tasks were delegated to any Mesos slave nodes that did not satisfy the constraint requirements. While the solution does segment the tasks as defined, it may not be sufficient for use cases where strict compliance and regulation is the motivation for segmenting data.

However, data segmentation may be interesting when considering task locality, as it may be desired to place certain services in a particular data center for some reason other than privacy reasons. For example, it may be desired to group a number of tasks together for performance reasons. Segmenting for this purpose is possible and should be adequate with the use of Mesos framework Marathon constraints feature.

However, this may not be good enough for every use case that requires segmentation of data, as there may be strict regulations and compliance to satisfy. It could be a requirement of the data not crossing the border, to guarantee that the data is not exported outside a nation. In order to properly guarantee data segmentation, source code analysis coupled with auditing tools to verify and attest that the data segmentation is functioning as intended is required. Alternatively a self developed framework that works by using the primitives provided by Apache Mesos. This is possible due to the fact that it is built as a two-level scheduler, allowing the individual frameworks to implement the algorithm for deciding whether or not to accept or reject offered resources based on the attributes and resources offered. A framework can therefore be designed to focus on data segmentation with the required functionality and auditing tools to guarantee and attest for compliance.

In the end, if security is of the highest order, then the hybrid cloud introduces additional vectors of attacks and this is something that needs to be considered. Attack vectors increase with the use of publicly accessible services. VPN does isolate the traffic in transit, however the public cloud providers have direct access to the virtual resources and by extension the data contained within it. The terms of services for each provider may contain clauses that gives them ownership of the data stored or the right to disclose the information to third parties. These are aspects of public cloud providers that needs to be considered before using them with sensitive type of data.

7.1.4 Cloud bursting and spot price instances

How can one automate the use of spot price instances to accommodate for cloud bursting?

The cloud bursting solution outlined in this thesis automatically deploy additional resources in the public cloud when the resource usage exceed a presepcified threshold. Additionally, the solution utilizes spot price instances and calculates a bidding price that is based on the current market price and a preset padding to ensure a winning bid.

The solution scales of a prespecified burst point threshold that is set in the configuration file. This solution is therefore quite rudimentary and does not account for edge cases as it will only read the usage percentage and scale based on this. The reason for why such a basic scaling decision algorithm was picked, was due to the lack of appropriate information to scale the solution on. The Marathon framework for Mesos does provide information about the tasks that is queued and await deployment however, the API does not provide the reason for why tasks are queued. This means that the API does not distinguish between tasks that are queued up because of resource depletion or tasks that can not be deployed due to constraints requirements.

Additionally, in a multi-framework environment, where Hadoop, Chronos, Marathon, or more frameworks are running simultaneously, the script would have to account for the resource usage between all the participating frameworks. This complicates the scaling decision algorithm. This is the reason for why the solution scales up in steps and not in one big burst, as there are no information that can be used to accurately gauge the needed resources, and therefore the needed number of spot instance slave nodes. A possible solution for this is to make an algorithm that evaluates the state of the Mesos cluster in combination of the information given by the frameworks as a composition and make a decision. This requires considerably complex logic to achieve with the information given by the APIs.

The pricing algorithm implemented is also quite simple and will bid the *current price* + x, where x is a prespecified padding set in the configuration. There are several other bidding strategies to which one can build a pricing algorithm as mentioned in Results: Design chapter 4. For the price bidding strategy to be viable, it has to win a bid within a reasonable amount of time, as the resources are needed in when the bids are made. Additionally, for the bid to be cost efficient, the proposed algorithm also has to consider on-demand prices.

While the solution does not account for every possible use case, it proves the viability of a cloud bursting solution using spot instances on a hybrid cloud platform using Apache Mesos.

7.2 Other limitations and considerations

7.2.1 Performance and abstraction

In this thesis, performance has not been evaluated for the hybrid cloud platform. Except from the feasibility of the hybrid cloud setup, performance is perhaps the most important factor to consider when attempting to create such a platform. Due to the way the hybrid cloud platforms may be set up, increased latencies are incurred. The effect of the increased latencies for Apache Mesos is unknown. As mentioned in the Approach chapter 3, Apache Mesos benefits from fast networking and latencies observed out on the Internet may possibly have an adverse effect on the performance of the cluster.

The prototypes implemented in this thesis uses multiple levels of abstraction that each incur some overhead. Hardware resources were abstracted through the use of virtualization due to how Altocloud and Amazon Web Services EC2. If performance is important, one should consider installing

Apache Mesos in on bare-metal hardware. The network was abstracted and secured with VPN, which also entails some overhead. However, it could argued that the overhead is required, as VPN also secures the traffic. Finally, there is the abstraction done by Apache Mesos. Apache Mesos abstracts the the machines, virtual or not, and presents them to applications and frameworks as a pool of available resources.

Additionally, the locations of the cloud sites are important to consider as well when performance is vital. Depending on the target audience and performance requirements, the location of the cloud sites could severely affect the performance of not only the cluster itself, but also perceived quality of service from the perspective of the users of the hybrid cloud.

Before any performance and latency sensitive workloads are considered for such a setup, performance and overheads should be evaluated.

7.2.2 Depth of testing and experiments

One limitation of this thesis is the lack of long-term experiments conducted on availability. Solutions were designed, evaluated, implemented, and analyzed during the thesis, however no verification beyond theory was done. As mentioned in the Approach chapter 3, it was assessed that to get any real indication of actual availability rates for the platform, uptime has to be monitored over a period of at least a year or more to be close to accurate. This is due to the nature of the metric tied to availability, as it is denoted with a percentage representing availability as function of time. For instance, at the time of writing, the downtime for Amazon Web Services EC2 for the availability region *eu-central-1* (Frankfurt) and *eu-west-1* (Ireland) had a total of 73 seconds of downtime for the last year (CloudHarmony, n.d.). Due to the high rates of availability it is difficult to gauge it in a short period of time. For an experiment lasting a few months it is possible that the availability rate found is 100%.

The opposite is also true. If a cloud provider is unlucky and experiences downtime during an experiment, the resulting availability rate may be terrible for the duration of the experiment, but could be impeccable for the rest of the year, resulting in a high availability rate. To properly gauge availability, a long-term experiment has to be conducted, preferably one to two years in duration at least.

Regarding the cloud bursting solution, a few experiments were conducted to verify the main functionality of the script. The main purpose of the experiments were to showcase two use cases that the script is intended to be used for, namely scaling up and down Mesos slave nodes depending on the needs for resources. While the experiments were sufficient to verify the feasibility of the proposed solution, the data were only collected for the few experiments outlined in the the Results: Implementation chapter 5, and lacks quantitative foundation to be used for something more than indication of the expected behaviour of the script. It is not given that the script will handle every use case as outlined in the results and further experiments must be conducted to verify that the behaviour is stable for other use cases.

7.2.3 DNS management

As no proper configuration and infrastructure was set up for resolving hostnames, there are several limitations with the testbed. For public facing applications, a proper DNS environment has to be in place for Apache Mesos and Marathon to utilize service discovery for serving publicly accessible services. With the *Mesos-DNS* project, public services can be made available to the Internet (Mesosphere, Inc., n.d.-a) by dynamically binding hostnames to the tasks. However, this requires the Mesos slave nodes that hosts those tasks to have Internet routable IP-addresses. Otherwise, a load balancer like HAProxy or Nginx has to be set up.

7.2.4 Spot price instances

Currently, there are no other providers of the spot price model for instances. The solution developed for automated cloud bursting may not work for other spot price model providers should they emerge, as the principles and the mechanisms may differ considerably. The cloud bursting script is therefore tailored for Amazon Web Services due to lack of alternative spot price markets.

An interesting observation during the cloud bursting experiments, were the volatility the prices for certain instance types. With fewer than 10 spot requests, the script were able to push the price upwards from 0.124 at an interval of 0.010 up until the experiment was terminated with price hitting 0.214 before dropping down to 0.124 just a few minutes later. Due to such a volatility, alternative instance types, availability regions, instance billing types, or an another cloud provider should considered when instance stability is important.

7.2.5 OpenNebula and virtualization

While OpenNebula may have been an adequate solution, it is overly complex for workloads where a simple, but powerful data center is desired, as OpenNebula is first and foremost a cloud manager. This entails higher complexity in the code with multiple services and abstraction to facilitate a cloud environment. The rich feature set increases the probability of experiencing issues that affect availability, as there are many parts that inter-operate. Apache Mesos is in comparison easier to install and manage and incurs less overhead, as resource isolation with *cgroups* or Docker replaces virtualization. OpenNebula revolves around virtual machines, providing flexibility and features with this prioritizing. On the other hand Apache Mesos puts applications, tasks, and services in the center and provides flexibility at application level, which for a larger scale data center is arguably more important, as it is ultimately the applications and services that provides utility.

Additionally, with virtualization, there are arguably two main advantages among many that drives the motivation for using virtual machines in a data center scenario. *Isolation* or commonly referred to as sandboxing, for increased security and to enable multi-tenancy use cases. And the second one being *resource flexibility and utilization* for increasing resource utilization per virtual machine, as they can be migrated for optimal resource usage on the physical hardware. Additionally, the flexibility of resource partitioning, enabling system administrators to shrink, expand, and relocate virtual machines depending on the requirements.

Docker, an emerging and increasingly popular containerization technology, can for certain workloads be a better alternative than full fledged virtual machine sandboxing as it has a lighter footprint. Combined with Apache Mesos which facilitates application level flexible resource partitioning, the need for virtualization technology diminishes for data center scenarios.

7.2.6 Apache Mesos

Apache Mesos is a relatively new piece of technology with the current version being 0.22.1. Although Apache Mesos is used in production environments in large companies like Twitter and Airbnb, there is a possibility of significant changes to the API and how Apache Mesos functions.

As mentioned in the Background chapter 3, Apache Mesos currently only support Linux environments. For applications requiring other operating systems, Apache Mesos is not the choice for now. If there is large enough demand for it, support may arguably be implemented in a future version.

7.2.7 Suitability

It is important to consider the the requirements of the use case for a setup of this size. For hosting a small personal website, installing and managing Apache Mesos may be overkill and result in far more complexity that can not easily be justified. However, there are a situations where it may be appropriate to use Apache Mesos, even for low requirement use cases. It may be for educational purposes or to dimension for future scalability, as Apache Mesos is easily scalable after setup.

For existing data centers, it may be difficult to migrate to an Apache Mesos cluster due to legacy software with old dependencies or non-Linux applications. It is important to consider the cost of migration in both time and resources compared to the potential benefits. Migrating existing solutions may be a large undertaking depending on the size and complexity.

7.3 Future work and improvement suggestions

This section contains suggestions for improvements and further work that would interesting to investigate.

7.3.1 Evaluating the performance and long-time availability of a hybrid cloud setup

In this thesis, a hybrid cloud solution with automated cloud bursting has been designed and implemented. With a working setup, what remains to be done is to gauge the long-time availability and performance of the proposed prototypes. The results of those investigations in combination with this thesis would make it possible to determine the level of utility the proposed solutions has.

Another aspect that would be interesting to investigate is QoS. This has to be gauged by monitoring the services running on the hybrid cloud platform and comparing it to the expected quality of service. It could be the expected response time, latency, performance, or any other metric that affect quality of the service.

7.3.2 Evaluate and prototype additional public cloud providers for a hybrid cloud platform

While the different availability zones for Amazon Web Services functioned as different cloud locations in the thesis, it does not consider any additional public cloud providers other than Amazon Web Services. Other cloud providers may use different architectures, locations, hypervisors, and hardware among other things that in theory should increase the availability if used in the hybrid cloud platform, as the degree of independence increases. Though, for the proposed solution it requires the other public cloud providers to facilitate the setup of a site-to-site VPN solution.

An evaluation of the possibilities of using additional public cloud providers to build a hybrid cloud platform would be interesting to investigate further.

7.3.3 Improve the cloud bursting solution

There are many interesting improvements that may be done to the automated cloud bursting script. Regarding the price bidding algorithm for spot instances, a new algorithm should be developed that is far more resistant to price spikes and that also considers on-demand prices. For instance, the script may chose to boot up a on-demand instance instead of a spot price instance when the spot prices exceeds the on-demand prices.

Another improvement to the script is to develop a new or expand the existing scale decision logic to rely on more tangible data when a scaling decision is made. By finding more suitable data to aid in the decision to scale, the script could more intelligently request the required number of nodes directly instead of scaling one node per script iteration.

Sometimes a lower tier instance-type is more expensive than a higher tier instance type solely because of demand. This was observed during this thesis. An algorithm could therefore be developed, which picks the most price efficient instance type based on the spot market prices for each instance type normalized with respect to the amount of resources on the virtual machine. With such an algorithm, the most price efficient instance type for the required amount of resources can be requested without being limited to one instance type. A thesis written by Borgenholt (2013) focuses on such a concept and uses a theater analogy to evaluate and pick a suitable virtual machine given certain performance and pricing requirements.

Although there are no other spot price market competitors among the other public cloud providers other than Amazon Web Services, the pricing for the on-demand instances differ. While these prices are fairly static, the cloud bursting solution could evaluate the pricing for other cloud providers as well and pick the most suitable option depending on the current availability and resource needs.

7.4 Potential impact of the thesis

Apache Mesos introduces a new way of thinking data center resources and may cause a change in paradigm regarding how data centers are built and managed. It is a proven concept with Twitter running Apache Mesos in their production environment for a few years now. Apache Mesos puts applications and services in the center and abstracts the layers below to provide a single unified interface to develop against. In this thesis, a hybrid cloud solution has been prototyped and implemented and challenges, potential pitfalls, and advantages with such a deployment model has been evaluated and discussed. While there are other ways to set up a hybrid cloud that facilitates cloud bursting, few if any of them can compare to the ease of installation and management, and flexibility of Apache Mesos.

If the performance and long-time availability of Apache Mesos is adequate, then the proposed prototypes and solutions presented in this thesis ought to be viable. With the hybrid cloud solutions outlined, higher levels of availability beyond Apache Mesos alone is possible. As for segmentation, it is be possible to segment certain applications and tasks that are not designed for high distribution to be located in the same rack or data center for increased performance, thus partly negating the latency overhead caused by the location of the individual cloud sites in the hybrid cloud platform.

The automated cloud bursting solution proposed sets the foundation for a cloud bursting solution using Apache Mesos and spot price instances to make an easily scalable solution. With some improvements, the solution represents a proof-of-concept in regards to cloud bursting using Apache Mesos.

In combination, prototype 2 with cloud bursting along with the segmentation seems to be to the a viable and practical solution that balances different aspects of a hybrid cloud setup and is most likely what one would expect of a hybrid cloud setup of an organization. This would arguably be due to larger interest for organizations with existing data centers that would like to simply extend workloads to the public cloud instead of offloading huge parts of the workload to public cloud providers.

At the time of writing, no documented attempts where discovered that attempts at deploying Apache Mesos using multiple sites of heterogeneous nature. Consequently, this thesis explores new concepts and solutions in regards to Apache Mesos and hybrid clouding. While the final solution ended up abstracting the network using VPN, a few discoveries were made in the attempts to deploy Apache Mesos without relying on network abstraction.

Chapter 8

Conclusion

This thesis presents multiple prototypes of setting up a hybrid cloud platform using Apache Mesos to weave together heterogeneous cloud types and geographical locations into a unified platform. The prototypes proposed each focuses on a specific perspective, maximizing availability and local access prioritization.

Data segmentation has also been demonstrated in this thesis with the hybrid cloud platform using Apache Mesos and the framework Marathon to set constraints to segment data flow. However, for strict requirements in terms of compliance or high level of confidentiality requirements, the outlined solution may not be adequate. Alternative solutions has been proposed.

An automated cloud bursting solution has also been prototyped and implemented on the hybrid cloud platform. It automatically requests spot instances in Amazon Web Services EC2 depending on resource usage of the hybrid cloud platform. The solution calculates a bidding price deviated from the current market price and adapts to price fluctuations. When resource usage decreases, the solution considers billing mechanics to ensure that resources that has been paid for are be fully utilized for maximum economical efficiency before terminating excessive resources.

While the thesis presents functional and viable solutions with respect to availability, segmentation and automated cloud bursting for a hybrid cloud platform, further work remains to improve and confirm the proposed solution, in particular a performance analysis of the proposed solutions.

Bibliography

Agmon Ben-Yehuda, O., Ben-Yehuda, M., Schuster, A., & Tsafrir, D. (2013). Deconstructing amazon ec2 spot instance pricing. *ACM Transactions on Economics and Computation*, *1*(3), 16.

alexr_. (2015, April). Displaying #mesos/2015-04-13.log. Retrieved April 19, 2015, from http://wilderness.apache.org/channels/?f=mesos/2015-04-13

Amazon Web Services, Inc. (n.d.-a). Amazon ec2 spot instances. Retrieved April 19, 2015, from http://aws.amazon.com/ec2/purchasing-options/spot-instances

Amazon Web Services, Inc. (n.d.-b). Aws | high performance computing. Retrieved March 19, 2015, from http://aws.amazon.com/hpc

Apache Software Foundation. (n.d.-a). C client bug in zookeeper_init (if bad hostname is given). Retrieved May 4, 2015, from https://issues.apache.org/jira/browse/ZOOKEEPER-1029

Apache Software Foundation. (n.d.-b). Mesos crashes if any configured zookeeper does not resolve. Retrieved May 4, 2015, from https://issues.apache.org/jira/browse/MESOS-2186

Apprenda Inc. (n.d.). Hybrid cloud. Retrieved March 16, 2015, from http://apprenda.com/library/glossary/hybrid-clouds-a-definition

Bicer, T., Chiu, D., & Agrawal, G. (2011, September). A framework for data-intensive computing with cloud bursting. In *Cluster computing (cluster), 2011 ieee international conference on* (pp. 169–177). doi:10.1109/CLUSTER.2011.21

Bittman, T. (2012, September). Mind the gap: here comes hybrid cloud. Gartner Blog Network. Retrieved April 19, 2015, from http://blogs.gartner.com/thomas_bittman/2012/09/24/mind-the-gap-here-comes-hybrid-cloud

Borgenholt, G. (2013). Audition: a devops-oriented quality control and testing framework for cloud environments.

Breiter, G. & Naik, V. (2013, March). A framework for controlling and managing hybrid cloud service integration. In *Cloud engineering (ic2e), 2013 ieee international conference on* (pp. 217–224). doi:10.1109/IC2E.2013.48

Butler, B. (2015, January). Re-examining cisco's intercloud strategy. Network World, Inc. Retrieved March 17, 2015, from http://www.networkworld.com/article/2864857/cloud-computing/re-examining-cisco-s-intercloud-strategy.html

CloudHarmony. (n.d.). Cloudsquare service status. Retrieved May 14, 2015, from https://cloudharmony.com/status-1year-of-compute-group-provider

Ghodsi, A., Hindman, B., Konwinski, A., & Zaharia, M. (2010). Mesosproposal. Retrieved March 18, 2015, from http://wiki.apache.org/incubator/MesosProposal

Google Inc. (2015, February). Google trends - web search interest: multi cloud, hybrid cloud, inter cloud - worldwide, jan 2008 - jan 2015. Retrieved February 3, 2015, from http://www.google.com/trends/explore?hl=en-US#q=multi%20cloud,%20hybrid%20cloud,%20inter%20cloud&date=1/2008%2085m&cmpt=q

Hindman, B., Konwinski, A., Zaharia, M., Ghodsi, A., Joseph, A. D., Katz, R. H., ... Stoica, I. (2011). Mesos: a platform for fine-grained resource sharing in the data center. In *Nsdi* (Vol. 11, pp. 22–22).

IBM. (n.d.). Private and hybrid cloud. Retrieved March 17, 2015, from http://www.ibm.com/cloud-computing/uk/en/private-cloud.html

Interoute Communications Limited. (n.d.). What is a hybrid cloud? Retrieved March 16, 2015, from http://www.interoute.com/cloud-article/what-hybrid-cloud

Iosup, A., Ostermann, S., Yigitbasi, M., Prodan, R., Fahringer, T., & Epema, D. (2011, June). Performance analysis of cloud computing services for many-tasks scientific computing. *Parallel and Distributed Systems, IEEE Transactions on*, *22*(6), 931–945. doi:10.1109/TPDS.2011.66

Jayaram, K. R., Safford, D., Sharma, U., Naik, V., Pendarakis, D., & Tao, S. (2014). Trustworthy geographically fenced hybrid clouds. In *Proceedings of the 15th international middleware conference* (pp. 37–48). Middleware '14. Bordeaux, France: ACM. doi:10.1145/2663165.2666091

Leopold, G. (2015, February). Apache mesos emerges as datacenter os. enterprisetech.com, EnterpriseTech. Retrieved April 2, 2015, from http://www.enterprisetech.com/2015/02/23/apache-mesos-emerges-as-datacenter-os

Mell, P. & Grance, T. (2011, September). The nist definition of cloud computing. National Institute of Standards and Technology. Retrieved April 19, 2015, from http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf

Mesos. (n.d.). Myriad. Retrieved May 16, 2015, from https://github.com/mesos/myriad

Mesosphere, Inc. (n.d.-a). Mesos-dns. Retrieved May 9, 2015, from http://mesosphere.github.io/mesos-dns

Mesosphere, Inc. (n.d.-b). The mesosphere team. Retrieved March 18, 2015, from http://mesosphere.com/team

Metz, C. (2015). Return of the borg: how twitter rebuilt google's secret weapon. WIRED. Retrieved March 16, 2015, from http://www.wired.com/2013/03/google-borg-twitter-mesos

MODAClouds. (n.d.). Modaclouds. Retrieved March 16, 2015, from http://www.modaclouds.eu

Moreno-Vozmediano, R., Montero, R., & Llorente, I. (2011, June). Multicloud deployment of computing clusters for loosely coupled mtc applications. *Parallel and Distributed Systems, IEEE Transactions on*, *22*(6), 924–930. doi:10.1109/TPDS.2010.186

Nair, S., Porwal, S., Dimitrakos, T., Ferrer, A., Tordsson, J., Sharif, T., … Khan, A. (2010, December). Towards secure cloud bursting, brokerage and aggregation. In *Web services (ecows), 2010 ieee 8th european conference on* (pp. 189–196). doi:10.1109/ECOWS.2010.33

OpenNebula Projec. (n.d.). Opennebula. Retrieved May 17, 2015, from http://opennebula.org

PaaSage. (n.d.). Paasage: model-based cloud platform upperware. Retrieved March 17, 2015, from http://www.paasage.eu

Rackspace, Inc. (n.d.). Hybrid cloud computing, hybrid hosting by rackspace. Retrieved March 17, 2015, from http://www.rackspace.com/cloud/hybrid

Sanders, J. (2014, July). Hybrid cloud: what it is, why it matters. ZDNet. Retrieved March 16, 2015, from http://www.zdnet.com/article/hybrid-cloud-what-it-is-why-it-matters

Schwarzkopf, M., Konwinski, A., Abd-El-Malek, M., & Wilkes, J. (2013). Omega: flexible, scalable schedulers for large compute clusters. In *Sigops european conference on computer systems (eurosys)* (pp. 351–364). Prague, Czech Republic. Retrieved from http://eurosys2013.tudos.org/wp-content/uploads/2013/paper/Schwarzkopf.pdf

Shado, S. (2015, January). Azure trails aws & google in 2014 uptime numbers. Retrieved February 3, 2015, from http://www.cloudwedge.com/azure-trails-aws-and-google-in-2014-uptime-numbers-2001

Silasi, S. (2014, November). Amazon cloudfront – about today's downtime. Retrieved February 3, 2015, from http://mo.nitor.me/amazon-cloudfront-about-todays-downtime

The Apache Software Foundation. (2015a, March). Add ssl support to mesos. Retrieved March 1, 2015, from https://issues.apache.org/jira/browse/MESOS-910

The Apache Software Foundation. (2015b). Apache mesos. Retrieved March 16, 2015, from http://mesos.apache.org

The Apache Software Foundation. (2015c, February). Powered by mesos. Retrieved March 1, 2015, from http://mesos.apache.org/documentation/latest/powered-by-mesos/

Twitter, Inc. (2013, July). Mesos graduates from apache incubation. Retrieved March 1, 2015, from https://blog.twitter.com/2013/mesos-graduates-from-apache-incubation

UC Berkeley AMPLab. (2012, September). Managing twitter clusters with mesos - benjamin hindman. [Video file]. Retrieved March 6, 2015, from https://www.youtube.com/watch?v=37OMbAjnJn0

Verizon Enterprise Solutions. (2014). *State of the Market, Enterprise Cloud 2014*. Retrieved February 3, 2015, from http://cloud.verizon.com/enterprise-cloud-report

VMWare, Inc. (n.d.-a). Cloud computing. Retrieved March 17, 2015, from http://www.vmware.com/cloud-computing/hybrid-cloud.html

VMWare, Inc. (n.d.-b). Vrealize suite. Retrieved March 17, 2015, from http://www.vmware.com/products/vrealize-suite/features.html

Voorsluys, W. & Buyya, R. (2012, March). Reliable provisioning of spot instances for compute-intensive applications. In *Advanced information networking and applications (aina), 2012 ieee 26th international conference on* (pp. 542–549). doi:10.1109/AINA.2012.106

Wilkes, J. (2014, July 14). Cluster management at google. Faculty Summit at Google. Retrieved from http://static.googleusercontent.com/media/research.google.com/no//university/relations/facultysummit2011/2011_faculty_summit_omega_wilkes.pdf

Zachariassen, E. (2015, February). Kobler regnekraft mellom flere nettskyer. digi.no, Teknisk Ukeblad Media AS. Retrieved March 17, 2015, from http://www.digi.no/932831/kobler-regnekraft-mellom-flere-nettskyer

Zaharia, M., Hindman, B., Konwinski, A., Ghodsi, A., Joesph, A. D., Katz, R., ... Stoica, I. (2011). The datacenter needs an operating system. In *Proceedings of the 3rd usenix conference on hot topics in cloud computing* (pp. 17–17). USENIX Association.

Appendices

Appendix A

Install notes for Apache Mesos master nodes

Install notes derived from https://docs.mesosphere.com/getting-started/datacenter/install/

To prevent a bug/glitch/oddity of having the ZooKeeper electing new masters every minute, add the local hostname into the `/etc/hostsfile`. Use the public/elastic ip. Like this:

```
127.0.0.1 localhost
128.39.121.22 master1
```

Proceed with the install:

```
sudo apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF
DISTRO=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
CODENAME=$(lsb_release -cs)

echo "deb http://repos.mesosphere.io/${DISTRO} ${CODENAME} main" | \
  sudo tee /etc/apt/sources.list.d/mesosphere.list
sudo apt-get -y update

sudo apt-get -y install mesos marathon
```

Change `/etc/zookeeper/conf/myid` into an unique id (Match the hostname).

Add the following values in `/etc/zookeeper/conf/zoo.cfg`:

```
server.1=10.0.19.5:2888:3888
server.2=192.168.0.5:2888:3888
server.3=172.16.0.5:2888:3888
```

or

```
server.1=10.0.19.5:2888:3888
server.2=192.168.0.5:2888:3888
server.3=172.16.0.5:2888:3888
server.4=10.0.19.16:2888:3888
server.5=10.0.19.17:2888:3888
```

Restart ZooKeeper: `sudo service zookeeper restart`.

Change `/etc/mesos/zk` into:

```
zk://10.0.19.5:2181,192.168.0.5:2181,172.16.0.5:2181/mesos
```

or

```
zk://10.0.19.5:2181,192.168.0.5:2181,172.16.0.5:2181,10.0.19.16:2181,
 ↪   10.0.19.17:2181/mesos
```

Change `/etc/mesos-master/quorum` to:

```
2
```

or (for five masters)

```
3
```

Set `/etc/mesos-master/hostname` to a resolvable address.

Create a folder tree for Marathon with the command `mkdir -p /etc/marathon/conf` and put the same thing as above into `/etc/marathon/conf/hostname`.

```
sudo service mesos-slave stop
sudo sh -c "echo manual > /etc/init/mesos-slave.override"

sudo service mesos-master restart
sudo service marathon restart
```

Extra settings to set:

`/etc/mesos-master/cluster`: Name of the cluster

```
NoxCluster
```

/etc/marathon/conf/http_port: For storing information in ZooKeeper for the Marathon framework

```
zk://10.0.19.5:2181,192.168.0.5:2181,172.16.0.5:2181/mesos
```

or

```
zk://10.0.19.5:2181,192.168.0.5:2181,172.16.0.5:2181,10.0.19.16:2181,
 ↪  10.0.19.17:2181/marathon
```

/etc/marathon/conf/master: For choosing the Mesos-master to use for the Marathon framework

```
zk://10.0.19.5:2181,192.168.0.5:2181,172.16.0.5:2181/mesos
```

or

```
zk://10.0.19.5:2181,192.168.0.5:2181,172.16.0.5:2181,10.0.19.16:2181,
 ↪  10.0.19.17:2181/mesos
```

/etc/marathon/conf/http_port: For choosing the service port for the Marathon GUI

```
6060
```

Appendix B

Script for bootstrapping Apache Mesos slave nodes

```
#!/bin/bash

# Set some variables:
INIT_ZOOKEEPER_MESOS_URL=zk://10.0.19.5:2181,192.168.0.5:2181,
  ↪  172.16.0.5:2181,10.0.19.16:2181,10.0.19.17:2181/mesos
INIT_PRIVATE_IP=`curl -s http://169.254.169.254/latest/meta-data/local-ipv4`
INIT_HOSTNAME=`curl -s http://169.254.169.254/latest/meta-data/hostname |
  ↪  cut -d. -f1 `

# Add an entry in /etc/hosts to deal with a Mesos/ZooKeeper bug:
## https://issues.apache.org/jira/browse/MESOS-2186
grep -q "$INIT_PRIVATE_IP $INIT_HOSTNAME" /etc/hosts
if [ $? != 0 ]; then
        echo -e "\n#Private IP\n$INIT_PRIVATE_IP $INIT_HOSTNAME" >>
          ↪  /etc/hosts
fi

# Unzip
apt-get install unzip -y

#############################
# Apache Mesos
#############################
# Add the Mesosphere repository and install Mesos
apt-key adv --keyserver keyserver.ubuntu.com --recv E56151BF
DISTRO=$(lsb_release -is | tr '[:upper:]' '[:lower:]')
CODENAME=$(lsb_release -cs)
echo "deb http://repos.mesosphere.io/${DISTRO} ${CODENAME} main" | \
  tee /etc/apt/sources.list.d/mesosphere.list
apt-get -y update
apt-get -y install mesos
```

```

# Stop the ZooKeeper and Mesos-master service if it is running
sudo service zookeeper stop
sudo sh -c "echo manual > /etc/init/zookeeper.override"
sudo service mesos-master stop
sudo sh -c "echo manual > /etc/init/mesos-master.override"

# Set the ZooKeeper url for Mesos to use
echo $INIT_ZOOKEEPER_MESOS_URL > /etc/mesos/zk

# Configure some settings for Mesos and Marathon to use
echo $INIT_PRIVATE_IP > /etc/mesos-slave/hostname
mkdir -p /etc/marathon/conf
echo $INIT_PRIVATE_IP > /etc/marathon/conf/hostname

# Start/restart the Mesos-slave service
service mesos-slave restart

#############################
# Python pip
#############################
# Install pip
apt-get -y install python-pip

# Install virtualenv
pip install virtualenv

#############################
# Monit
#############################
# Install Monit
apt-get -y install monit

# Configure the check
cat > /etc/monit/conf.d/mesos-slave <<- EOM
check process mesos-slave
  matching "/usr/sbin/mesos-slave"
  start program = "/usr/sbin/service mesos-slave start"  with timeout 60
   ↪  seconds
  stop program = "/usr/sbin/service mesos-slave stop"
EOM

# Reload Monit
monit reload
```

Appendix C

Cloud bursting script

Example configuration and dependencies can be found at https://git.cs.hioa.no/s171636/burst.

```
#!/usr/bin/env python
import basics
import boto3
import yaml
import syslog
import getopt
import signal
import sys
import time
import datetime
import dateutil
import json
import urllib2
from math import ceil,floor

def print_usage():
    print "Automated cloud bursting script\n"
    print "usage: " + __file__ + " [arguments]\n"
    print "Arguments:"
    print "    --help\t\t\t Prints this help message"
    print "    -v [--verbose]\t\t Verbose output"
    print "    -c [--config]\t\t Specify another config file"

def get_params():
    try:
        opts, args = getopt.getopt(sys.argv[1:], "vc:",
         ↪   ["verbose","help","config="])
    except getopt.GetoptError as e:
        basics.handle_error(e)

    return opts, args

def set_options(opts):
    global verbose
    global config_path
    verbose = False
    config_path = False

    for o,p in opts:
        if o in ["-v", "--verbose"]:
            verbose = True
        elif o in ["--help"]:
            print_usage()
            exit()
        elif o in ["-c", "--config"]:
            config_path = p

def print_verbose(message):
    global verbose
    if verbose:
        print message

def sleep(start_timestamp, duration):
    loop_time = time.time() - start_timestamp
    sleep_time = duration - loop_time
    print_verbose("The script used " + str(loop_time) + " seconds this
     ↪  loop")
    if sleep_time <= 0:
        print_verbose("Time used >= the set interval. Skipping sleep.")
        sys.stdout.flush()
    else:
        print_verbose("Sleeping additional " + str(sleep_time) + " seconds")
        sys.stdout.flush()
        time.sleep(sleep_time)

def import_config():
    global config_path
    if config_path == False:
        config_path = 'config.yml'

    if not basics.check_file_exists(config_path):
        print_verbose('Attempted to find config file: %s' % config_path)
        basics.handle_error('No configuration file found')

    try:
        with open(config_path, 'r') as configfile:
            config = yaml.load(configfile)
    except StandardError as e:
        print_verbose(e)
        basics.handle_error('Error when attempting to read the configuration file')

    return config

def purge_old_spot_requests(ec2client,cur_spot_requests,timeout,max_bid):
    now_time = datetime.datetime.utcnow()
    now_time = now_time.replace(tzinfo=dateutil.tz.tzutc())

    for request in cur_spot_requests:
        if request[u'State'] == 'open':
            #print now_time - request[u'CreateTime']
            if (int(now_time.strftime('%s')) - int(request[u'CreateTime'].strftime('%s')) > timeout and
                not request[u'SpotPrice'] == max_bid):
                try:
                    print_verbose('Lingering spot request, canceling %s' % request[u'SpotInstanceRequestId'])
                    response = ec2client.cancel_spot_instance_requests(SpotInstanceRequestIds = [request[u'SpotInstanceRequestId']])
                except Exception as e:
                    print_verbose(e)
                    basics.handle_error('Some error ocurred. Could not cancel old spot instance request.')


def get_current_spot_slaves(ec2resource):
    spot_slaves = []
    filter = [{
                'Name': 'instance-lifecycle',
                'Values': ['spot'],
              },
              {
                'Name': 'instance-state-name',
                'Values': ['pending','running','rebooting']
              },
             ]

    for instance in ec2resource.instances.filter(Filters=filter):
        spot_slaves.append(instance)

    return spot_slaves,ec2resource.instances.filter(Filter=filter)

def get_current_spot_requests(ec2client,states):
    spot_requests = []

    data = ec2client.describe_spot_instance_requests()['SpotInstanceRequests']
    if states == 'all':
        return data

    for request in data:
        if request['State'] in states:
            spot_requests.append(request['SpotInstanceRequestId'])

    return spot_requests

def import_launch_config():
    config_path = 'launch_config.yml'
    if not basics.check_file_exists(config_path):
        print_verbose('Attempted to find config file: %s' % config_path)
        basics.handle_error('No configuration file found')

    try:
        with open(config_path, 'r') as configfile:
            config = yaml.load(configfile)
    except StandardError as e:
        print_verbose(e)
        basics.handle_error('Error when attempting to read the configuration
         ↪  file')

    return config

def request_spot_instances(ec2client,num_to_boot,instance_type,max_bid):
    launch_config = import_launch_config()

    try:
        response = ec2client.request_spot_instances(SpotPrice=str(max_bid),
                            InstanceCount=num_to_boot,
                            LaunchSpecification=launch_config)
    except Exception as e:
        print_verbose(e)
        basics.handle_error('Requesting spot instances failed')

def cancel_spot_requests(ec2client,spot_requests,num_to_cancel):
    try:
        response = ec2client.cancel_spot_instance_requests(
         ↪  SpotInstanceRequestIds=spot_requests[-num_to_cancel:])
    except Exception as e:
        print_verbose(e)
        basics.handle_error('Termination of spot requests failed')

def terminate_spot_instances(ec2client, spot_instances, raw_spot_info,
 ↪  num_to_terminate, partial_hour_limit):
    try:
        list_with_timestamp = dict()
        now_time = datetime.datetime.utcnow()
        now_time = now_time.replace(tzinfo=dateutil.tz.tzutc())
        terminated = 0

        for instance in spot_instances:
            list_with_timestamp[instance.instance_id] =
             ↪  instance.launch_time.strftime('%s')

        for instance in sorted(list_with_timestamp,
         ↪  key=list_with_timestamp.get):
            if terminated == num_to_terminate:
                break

            instance_lifetime_delta = int(now_time.strftime('%s')) -
             ↪  int(list_with_timestamp[instance])

            # Calculate minutes in a partial hour used
            if instance_lifetime_delta < partial_hour_limit:
                part_seconds = instance_lifetime_delta
            else:
                part_seconds = instance_lifetime_delta -
                 ↪  (floor(instance_lifetime_delta/3600)*partial_hour_limit)

            if part_seconds > partial_hour_limit:
                print_verbose('Terminating %s...' % instance)
                response =
                 ↪  ec2client.terminate_instances(InstanceIds=[instance])
            else:
                print_verbose('%s has not reached the set partial hour
                 ↪  limit. %.0f minutes has passed.' %
                 ↪  (instance,part_seconds/60))

            terminated += 1

    except Exception as e:
        print_verbose(e)
        basics.handle_error('Termination of spot requests failed')

def fetch_current_mesos_master(mesos_zkurl):
    print_verbose('Resolving ZooKeeper url for the working Mesos Master')

    #return '52.17.132.212:5050'

    try:
        mesos_master = basics.run_command('mesos-resolve %s' % mesos_zkurl)
    except Exception as e:
        print_verbose(e)
        basics.handle_error('Could not resolve the ZooKeeper url for the
         ↪  leading master node.')

    return mesos_master

def fetch_and_parse_json(request):
    try:
        print_verbose('Fetching %s' % request)
        json_data = urllib2.urlopen(request).read()
        parsed_data = json.loads(json_data)
    except Exception as e:
        print_verbose(e)
        basics.handle_error('Failed JSON fetch and parse.')

    return parsed_data

def get_scaling_decision(resources_in_use, current_percent_in_use,
↪  active,pending, config):

    # Set some settings from the configration
    burst_point = config['burst_point_percentage']
    max_slaves = config['maximum_spot_slaves']

    baseline_cpus = config['baseline_cpus']
    baseline_mem = config['baseline_mem']
    baseline_disk = config['baseline_disk']

    instance_cpus = config['instance_cpus']
    instance_mem = config['instance_mem']
    instance_disk = config['instance_disk']

    pending_active_slaves = active + pending

    # Calculate pending resources
    pending_active_spot_resources = {'cpus': pending_active_slaves *
     ↪  instance_cpus,
                                     'mem': pending_active_slaves *
                                      ↪  instance_mem,
                                     'disk': pending_active_slaves *
                                      ↪  instance_disk}

    # Total of pending, active and baseline resources
    total_resources_apb = {'cpus': baseline_cpus +
     ↪  pending_active_spot_resources['cpus'],
                           'mem': baseline_mem +
                            ↪  pending_active_spot_resources['mem'],
                           'disk': baseline_disk +
                            ↪  pending_active_spot_resources['disk']}

    # Calculate usage percentage if the pending resources would have been
     ↪  available
```

```
pending_usage_percent = {'cpus':
 ↪  resources_in_use['cpus']/total_resources_apb['cpus'],
                'mem':
                 ↪  resources_in_use['mem']/total_resources_apb['mem'],
                'disk':
                 ↪  resources_in_use['disk']/total_resources_apb['disk']}

print_verbose('   |------------------------------------------')
print_verbose('   |       Burst point value set to %.2f      ' %
 ↪  burst_point)
print_verbose('   |------------------------------------------')
print_verbose('   |------------------------------------------')
print_verbose('   | Resource usage: |  Percent\t | Count      ')
print_verbose('   |------------------------------------------')
print_verbose('   | CPUs            |  %.2f%%\t | %.2f       ' %
 ↪  (current_percent_in_use['cpus']*100,resources_in_use['cpus']))
print_verbose('   | Memory          |  %.2f%%\t | %i MB      ' %
 ↪  (current_percent_in_use['mem']*100,resources_in_use['mem']))
print_verbose('   | Disk            |  %.2f%%\t | %i MB      ' %
 ↪  (current_percent_in_use['disk']*100,resources_in_use['disk']))
print_verbose('   |------------------------------------------')


# Scale up if the burst point is lower than the resources used
if any(pending_usage_percent[i] >= burst_point for i in
 ↪  pending_usage_percent):
    if pending_active_slaves + 1 > max_slaves:
        print_verbose('The specified limit for max number of slaves has
         ↪  been hit. Will not scale up.')
        return 0
    else:
        return 1

# Stop evaluating if the number of slaves is zero
if pending_active_slaves == 0:
    print_verbose('')
    return 0

# Calculate the number of slaves that can potentially be terminated
scale_down_num = 0
while pending_active_slaves >= 0:

    ## Calculate the usage percent if we remove scale_down_num slave
    x_less_usage = {
        'cpus': (resources_in_use['cpus'])/(total_resources_apb['cpus']
         ↪  - scale_down_num*instance_cpus),
        'mem': (resources_in_use['mem'])/(total_resources_apb['mem'] -
         ↪  scale_down_num*instance_mem),
            'disk': (resources_in_use['disk'])/(total_resources_apb['disk'] - scale_down_num*instance_disk)}

        # Set any negative values to zero
        for i in x_less_usage:
            if x_less_usage[i] < 0:
                x_less_usage[i] = 0.0

        # Check if the usage exceeds the burst point
        if not any(x_less_usage[i] >= burst_point for i in x_less_usage):
            scale_down_num += 1
            pending_active_slaves -= 1
            continue

        break

    # Subtract 1 from scale_down_num, otherwise the script will scale up and down
    # every minute. This will ensure that the number stays right above the the burst point
    scale_down_num -= 1

    if scale_down_num > 0:
        return - scale_down_num

    # No change
    return 0

def fetch_current_price(ec2client, avail_zone, instance_type, price_x, max_limit):

    # Fetch the most recent price entry for the set zone and instance type
    price_entry = ec2client.describe_spot_price_history(
                                  InstanceTypes=[instance_type],
                                  AvailabilityZone=avail_zone,
                                  ProductDescriptions=['Linux/UNIX (Amazon VPC)'],
                                  MaxResults=1)

    original_price = float(price_entry['SpotPriceHistory'][0][u'SpotPrice'])
    bid = float(price_entry['SpotPriceHistory'][0][u'SpotPrice']) + price_x


    print_verbose('   |--------------------------------')
    print_verbose('   | Curent market price:    %.3f   ' % original_price)
    print_verbose('   | Maximum bid limit       %.3f   ' % max_limit)

    final_bid = bid

    if bid > max_limit:
        final_bid = max_limit


    print_verbose('   | Our bid                 %.3f   ' % final_bid)
    print_verbose('   |--------------------------------')

    return final_bid

def main():
    # Get paramaters and process them
    opts, args = get_params()
    set_options(opts)

    # Import the configuration and set some session settings
    config = import_config()

    try:
        session = boto3.session.Session(
                        aws_access_key_id=config['aws_access_key_id'],

                         ↪   aws_secret_access_key=config['aws_secret_access_key'],
                        region_name=config['default_region'])
    except Exception as e:
        print_verbose(e)
        basics.handle_error('Could not establish a session towards AWS API,
         ↪   check config')

    # Start EC2 and resource and client session
    try:
        ec2resource = session.resource('ec2')
        ec2client = session.client('ec2')
    except Exception as e:
        print_verbose(e)
        basics.handle_error('Could not establish a session towards EC2.')

    # Main execution
    while True:
        start_time = time.time()
        print ''

        #################################
        ### Collect hybrid cloud metrics
        #################################

        ## Fetch current Mesos master
        mesos_master = fetch_current_mesos_master(config['mesos_zkurl'])

        ## Create a Marathon url
        marathon_url = '%s:%i' % (mesos_master[:-5],
         ↪  config['marathon_port'])
        print_verbose('   Current Mesos master %s' % mesos_master[:-5])

        ## Collect Mesos metrics
        mesos_data = fetch_and_parse_json('http://%s/metrics/snapshot' %
         ↪  mesos_master)

        ## Collect the usage values of the resources
        resources_in_use = {'cpus': float(mesos_data[u'master/cpus_used']),
                            'mem': float(mesos_data[u'master/mem_used']),
                            'disk': float(mesos_data[u'master/disk_used'])}
        current_percent_in_use = {
                            'cpus':
                             ↪  float(mesos_data[u'master/cpus_percent']),
                            'mem': float(mesos_data[u'master/mem_percent']),
                            'disk':
                             ↪  float(mesos_data[u'master/disk_percent'])}

        ### Collect EC2 metrics
        cur_slaves,cur_slaves_raw = get_current_spot_slaves(ec2resource)
        cur_spot_requests = get_current_spot_requests(ec2client,'all')
        cur_open_spot_requests =
         ↪  get_current_spot_requests(ec2client,[u'open'])
        num_active_pending_slaves = len(cur_slaves) +
         ↪  len(cur_open_spot_requests)

        ################################
        ### Calculate the bidding price
        ################################

        # Fetch the current price
        bid = fetch_current_price(ec2client,
                                  config['availability_zone'],
                                  config['instance_type'],
                                  config['price_x'],
                                  config['maximum_bid_limit']
                                  )

        ########################################################
        ### Make a descision of whether or not to cloud burst
        ########################################################
        slaves_to_adjust = get_scaling_decision(resources_in_use,
                                                current_percent_in_use,
                                                len(cur_slaves),
                                                len(cur_open_spot_requests),
                                                config)

        desired_slaves = len(cur_open_spot_requests) + len(cur_slaves) + slaves_to_adjust

        print_verbose('   |----------------------------')
        print_verbose('   | Number of:         | Count ')
        print_verbose('   |----------------------------')
        print_verbose('   | Desired instances  |   %i   ' % desired_slaves)
        print_verbose('   | Pending requests   |   %i   ' % len(cur_open_spot_requests))
        print_verbose('   | Active instances   |   %i   ' % len(cur_slaves))
        print_verbose('   |----------------------------')

        ############################
        ### Execute the descision
        ############################

        # Remove spot requests that exceeded the timeout and that does bid at max limit
        purge_old_spot_requests(ec2client,
                                cur_spot_requests,
                                config['spot_request_timeout'],
                                config['maximum_bid_limit'])

        ## The number of pending and active instances are ok
        if desired_slaves == num_active_pending_slaves:
            print_verbose('The number of pending and active slaves are ok')
            try:
                sleep(start_time,config['execution_interval'])
            except KeyError as e:
                basics.handle_error('%s has not been set in the config.' % e)

            continue

        ## Request new spot instances
        if desired_slaves > num_active_pending_slaves:
            print_verbose('Not enough pending or active slave nodes. Requesting new ones')
            request_spot_instances(ec2client,
                                   desired_slaves-num_active_pending_slaves,
                                   config['instance_type'],
                                   bid)

        ## Terminate excessive pending spot requests
        if (num_active_pending_slaves > desired_slaves
            and not len(cur_open_spot_requests) == 0):
            excessive_slaves = num_active_pending_slaves - desired_slaves

            if excessive_slaves < len(cur_open_spot_requests):
                print_verbose('Excessive spot requests. Attempting to cancel
                 ↪   %i' % excessive_slaves)
                cancel_spot_requests(ec2client, cur_open_spot_requests,
                 ↪   excessive_slaves)
                num_active_pending_slaves = num_active_pending_slaves -
                 ↪   excessive_slaves
            else:
                print_verbose('Excessive spot requests. Attempting to cancel
                 ↪   %i' % len(cur_open_spot_requests))
                cancel_spot_requests(ec2client, cur_open_spot_requests,
                 ↪   len(cur_open_spot_requests))
                num_active_pending_slaves = num_active_pending_slaves -
                 ↪   len(cur_open_spot_requests)

        ## Terminate excessive spot instances
        if (num_active_pending_slaves > desired_slaves):
            excessive_slaves = num_active_pending_slaves - desired_slaves
            print_verbose('Excessive spot instances. Attempting to terminate
             ↪   %i' % excessive_slaves)

            terminate_spot_instances(ec2client, cur_slaves, cur_slaves_raw,
             ↪   excessive_slaves, config['partial_hour_limit'])


        ### Sleep and repeat
        try:
            sleep(start_time,config['execution_interval'])
        except KeyError as e:
            basics.handle_error('%s has not been set in the config.' % e)

if __name__ == '__main__':
    original_sigint = signal.getsignal(signal.SIGINT)
    signal.signal(signal.SIGINT, basics.exit_script)
    main()
```