Using Heat and Ceilometer to create an elastic OpenStack grid

Niklas Trippler

Master's thesis in Software Engineering at

Department of Computing, Mathematics and Physics,

Bergen University College

Department of Informatics,

University of Bergen

March 2017

Preface

This is a master thesis in software engineering performed at the Grid group at Bergen University College from 2015 to 2017. The idea for this project came from Dario Berzano from the ALICE group at CERN, and has during the execution of the project been presented at ALICE offline week 2016 and ALICE Tier-1/Tier-2 workshop 2016 to align the goals of the project with the goals of ALICE.

I would like to thank Håvard Helstrup, and my supervisors Bjarte Kileng and Kristin Hetland for their great help, feedback and insight during this project.

Dario Berzano has a similar project, and I would like to thank him for the idea to use Heat and Ceilometer for this project.

I also want to thank the ALICE Offline group at CERN and the ALICE Tier1/Tier2 meeting for allowing me to present my ideas there and giving good feedback on the project. Additionally my fellow students Didrik A. Rognstad and Maxim Storetvedt also contributed with help and suggestions for the project.

Finally, I would like to thank my friends, family and Sunniva for all their support.

Bergen, 2017-03-16

Niklas Trippler

Niklas Trippler

Abstract

Grid computing is a term for connecting computing resources together to solve large computational problems. Computational grids are used for a lot of computations within the high energy physics domain, where the amount of computing power required for some tasks is vastly more than a local computer can provide.

This thesis investigates if cloud technology can be utilized to make an elastic computational grid, in order to get access to more resources that would otherwise be idle. Functional requirements were defined for creating a prototype capable of providing a virtualized environment that scales the amount of virtual machines up and down automatically based on the load on the system. A prototype was created to take advantage of the technology provided by cloud, and the prototype tested to see how it fulfills the functional requirements.

Although one of the functional requirements was not achieved, the test results demonstrate that the technology has promising potential, but further work and testing needs to be done.

Contents

List of Figures

Chapter 1

Introduction

Grid computing is a term for connecting heterogeneous computing resources together to solve large computational problems. A grid is a global platform with computing resources distributed at several sites at different geographical locations. Grid computing is used for a lot of computations within the high energy physics (HEP) domain, where the amount of computing power required is vastly more than a local data center can provide.

Grids today are usually static computational resources consisting of data centers from several administrative domains, with computers dedicated to running computational tasks.

Cloud computing is a concept that has become more popular recently. Cloud computing is a term for provisioning computing resources on-demand rather than constantly, and can have the ability to scale a service up or down automatically based on usage. Cloud services, like grids are run in large data centers, but where grid data centers are usually dedicated to running tasks for a group or a project, cloud data centers run many different isolated applications from many different users.

Cloud services make heavy use of virtualization, which is a technique for implementing a machine in software that runs an application or an operating system. Virtualization provides good isolation and security at the cost of some performance.

Grid is a technology that has been around since the early 2000s, and could potentially benefit from integration with the newer cloud computing concept. This thesis looks into how this can be achieved.

1.1 Problem Description

The amount of computing power required in a grid varies with time, and at times there can be more demand for computational power than the grid has available. This will cause tasks to take longer before they start, leaving users potentially waiting a long time for the results. Grid resources today are mainly static, and adding or removing compute resources in a grid to meet new computational demands takes a long time, potentially several days. Having the ability to scale the amount of computers in the grid up and down would enable the grid to take advantage of otherwise idle resources for periods of time until the resources are required for other tasks. This could be useful the when amount of jobs peak, as the grid could borrow resources to scale up in order to get the jobs processed faster, and then free up the borrowed resources when they are no longer needed or available. One could also scale beyond local resources by renting computational resources from cloud service providers. This can give the grid temporary access to additional resources when needed, which can be a lot cheaper than buying dedicated resources permanently.

Grid security is largely based on trust in the users of the grid to not run malicious code. Trusting users scales up to a point, but with more users the chances increases that one of them will do something malicious. If a malicious user was to run malware on the grid platform it could potentially stay there a long time and cause damage. Running the job in an isolated environment, like a virtual machine will provide increased security, as the job would no longer have direct access to the host. Setting up new nodes for job execution in a cloud is fast, and can be automatically done, potentially after each job, which would remove any remains from previous jobs.

Grid environments are heterogeneous, and differences in both hardware and software available on each node, may cause jobs to give different results, or even crash depending on which node it executes on. It is possible to define requirements for a job, and ensure that the worker that executes the job meets these. This may however result in some nodes that don't meet the particular set of requirements being left idle, while other nodes that have those requirements fulfilled get completely saturated and jobs start waiting for execution time, despite available computing resources.

As cloud computing relies on virtualization all machines running on a cloud solution can appear to have identical software and hardware to the appliance being executed. In this way cloud computing could also provide a homogeneous execution environment for the grid that is today composed of varying hardware and software as long as the computers can run a hypervisor[1]. This could remove cases where a job could get a different result based on which node it executed on due to for example differences in libraries being installed on the various nodes.

1.2 Objective

There are several approaches towards taking advantage advantage of newer technologies like cloud to get access to more resources for the grid. This thesis is one of the projects exploring how this can be done.

It is clear that grid computing can benefit greatly from some of the features provides by cloud. By using cloud technology, the grid can get access to more resources by scaling up and down the amount of computers, be more secure by isolating jobs from hosts, and provide a more stable environment by making the software homogeneous across all the nodes. This thesis will develop a prototype solution that will leverage cloud technologies to provide benefits to a grid and investigate how functional this solution is.

To summarize, the aim of this thesis is to build a prototype that provides a computational job execution environment, with automatic elasticity by leveraging cloud technology.

1.2.1 Functional Requirements

To narrow the scope of the research question, a number of functional requirements have been defined.

Job submission and execution The environment should be fully capable of executing grid jobs.

Recreatable environment In order to rapidly launch grid platforms either permanent or temporary, the system should be able to fully automatically launch, set up and configure a grid environment without any operator intervention.

[1]Hypervisor is defined more in depth in chapter 4

Automatic Scaling The system should automatically be able to detect when additional computational resources are required and scale up to meet those requirements, as well as scaling down when the resources are no longer required.

Manual control An operator should be able to control the amount of virtual machines and change the scaling on demand.

Job completion during down scaling If the system is removing worker nodes it should not remove the ones still executing jobs.

1.2.2 Reasoning for Criteria

Job submission and execution is one of the more obvious requirements as it is one of the defining characteristics of the problem space. **Recreatable environment** and **automatic scaling** are some of the features that cloud solutions can provide, and as the goal of the thesis is to leverage cloud features for job execution, those are important requirements for the measure of the success of the project.

Manual control is a requirement that, while not essential in a theoretical grid setting, is important in a real life setting where an operator may need to override some behavior or make some other changes to a running system.

As the system will be scaling up and down based on the load on the system it is important that worker nodes executing jobs will not be removed, as this could remove the results from a long-running job. For this reason the **job completion during down scaling** criteria was selected. It is not a critical criteria, as a job getting canceled because it's worker got removed would be resubmitted to be executed on another worker, but has the potential to waste immense amount of computing resources and time if a long-running job was canceled right before it completed.

Selecting a correct set of criteria for the evaluation of the project is a subjective task, and there are a lot of other criteria one could argue should be in the list. The criteria were selected to be a good balance of demonstrating which advantages and disadvantages such a solution has compared to a conventional grid site.

1.3 Thesis Overview

Chapter 2 provides a description of grid computing and some of the challenges related to this area.

Chapter 3 provides a description of cloud computing and the advantages it can bring to grid computing.

Chapter 4 provides a description of virtual machines and their use in the cloud setting as well as how they can improve security.

Chapter 5 provides an overview of other projects aiming to merge grid and cloud computing.

Chapter 6 describes the design of the prototype developed in this thesis as well as the design decisions that were made during development.

Chapter 7 describes the implementation of the prototype in depth, including some of the issues encountered and the workarounds implemented.

Chapter 8 is an evaluation of the solution and how it fulfills the requirements from 1.2, as well as the limitations of the validity of the results presented.

Chapter 9 is the conclusion of the project as well as suggestions for further work.

Chapter 2

Grid Computing

A grid is a global distributed platform where a user can send computational jobs that the user himself cannot run on local computing resources. The job can be too large, time consuming, or the user may lack some resource required for the job, that is available in the grid. The jobs the user sends into the grid get executed and the user can retrieve the result later, as shown in figure 2.1. A job in the grid context is a computational task that will be run and returns some result. Since the grid concept was introduced it has become one of the most important tools for handling computations that require large amounts of computational power or input/output operations. There are a lot of such computational demands when doing calculations on for example high energy physics and grid computing is a tool that is often used in this area.

2.1 Introduction

Ian Foster defines in *What is the Grid? A Three Point Checklist* [1] that a computational grid *coordinates resources that are not subject to centralized control using standard, open, general-purpose protocols and interfaces to deliver nontrivial qualities of service.* The fact that a grid uses standard and open general-purpose protocols and interfaces to operate is of particular importance as this allows interested parties to establish resource-sharing arrangements dynamically and allows geographically distributed interested parties to rapidly add and remove resources to new or existing grid infrastructures.

The purpose of grid computing is to use several geographically distributed connected com-

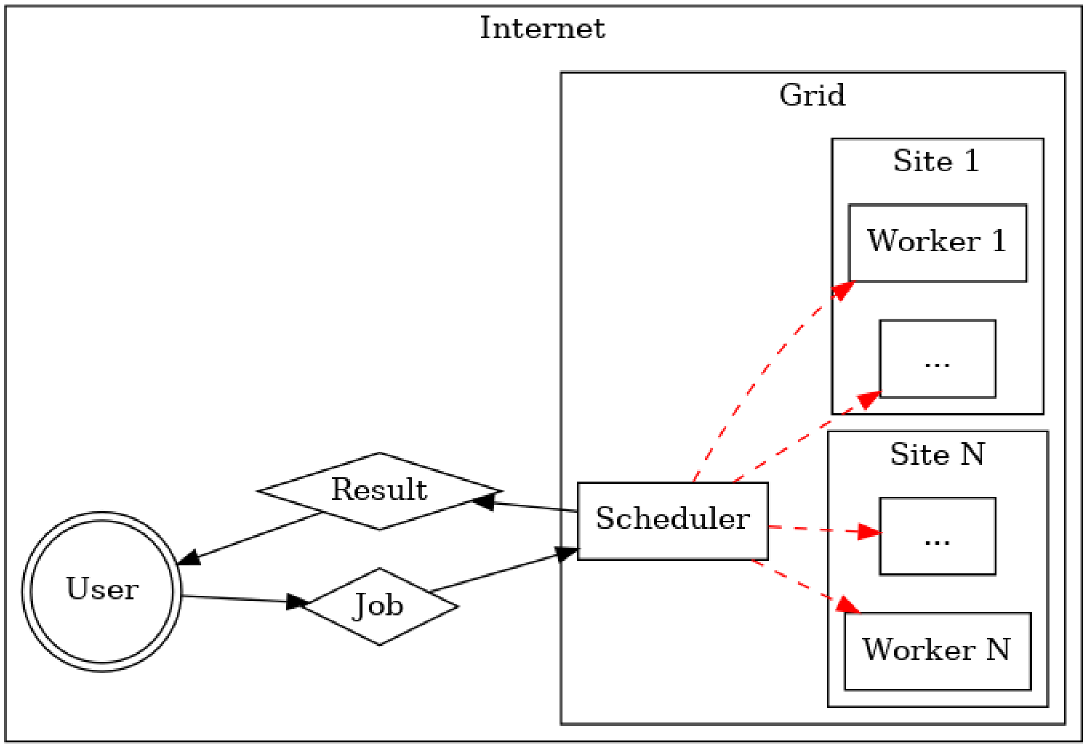

Figure 2.1: A user interacting with a grid.

putational sites that pool resources together to collectively provide adequate amounts of computing power. The term grid computing was popularized when Ian Foster and Carl Kesselman published *The Grid: Blueprint for a new computing infrastructure*[2] in 1999. Grids today usually run on several dedicated data centers from different administrative domains around the world.

With the introduction of high-speed network connections, distributed computer systems can more easily be connected. The connected systems can pool computers from multiple administrative domains to work towards a common goal, or to solve a single task. As a result of grid computing's distributed nature, both the hardware and software that makes up a grid platform is often heterogeneous with many different hardware configurations and software installations. Grid computing uses a grid middleware to give uniform access to the grid resources and distributing jobs among the available computational resources in the grid. Grid middleware will be discussed in greater detail in 2.2.

Since the various computational nodes are not under a single centralized control, available resources can appear and disappear, also in the middle of a computation, causing a job to fail.

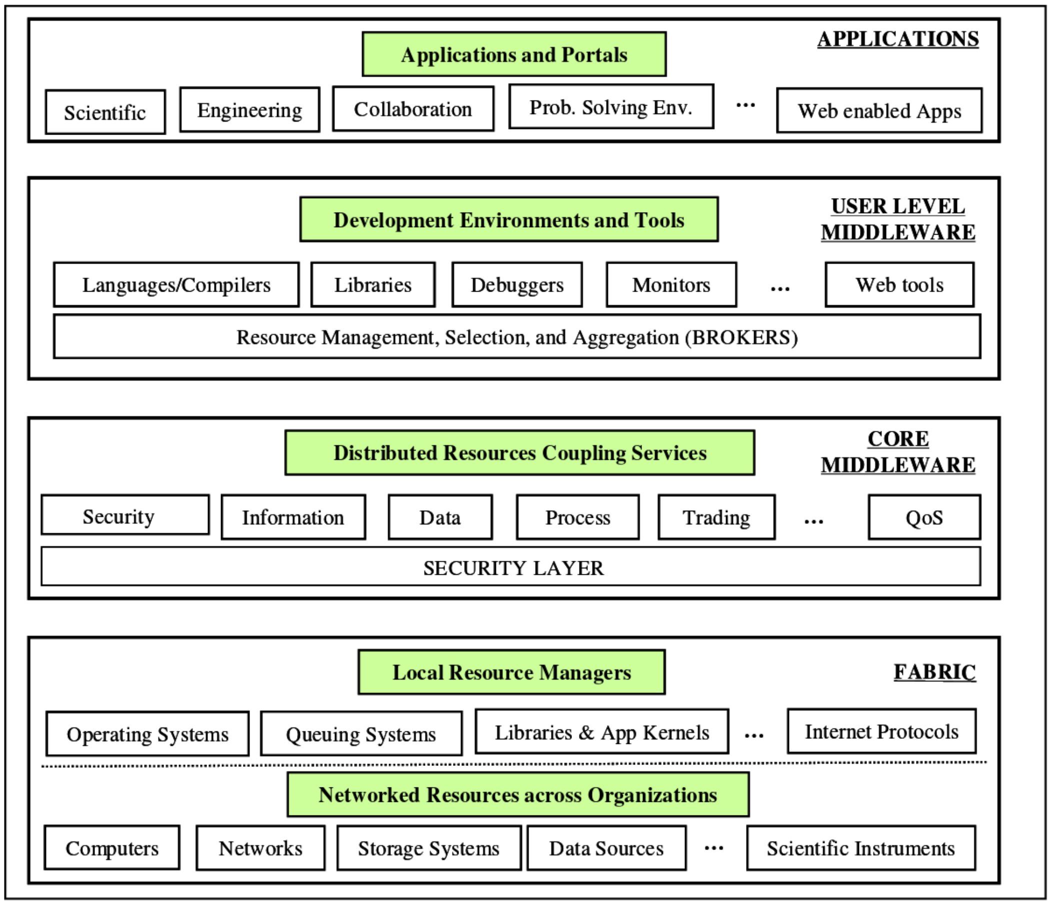

Figure 2.2: The different layers of grid middleware. Figure from [3].

Today's grids are fairly static, and entire data centers disappearing is a rare occurrence. Single nodes do on occasion disappear and the grid middleware needs to handle such exceptions and potentially resubmit a job that fails or times out. This, as well as ensuring proper access control over the Internet and efficiently managing available resources are some of the main challenges grid middleware needs to solve.

2.2 Middleware

As shown in figure 2.2 a grid consists of four layers: fabric, core middleware, user-level middleware and applications and portals layers[3]. In this thesis middleware refers to the user-level middleware.

As the computing resources in the grid are heterogeneous, middleware is used to provide a uniform way to access the various resources on the grid and provide development environments. Grid middleware is a collection of software that provide the infrastructure for submitting computational jobs to the Grid. Both security, resource-management, authentication mechanisms and APIs are provided by the grid middleware.

There are many types of grid middleware that can perform various services. A middleware could be as simple as a translator from the user-submitted job description file to the job description language used by the local batch system at the various sites to an advanced development environment that can split up a calculation into several jobs and submit them for the user. One such middleware is the Alice Environment (AliEn)[4], developed and used by the ALICE project at CERN.

2.3 Jobs

Early on in computing one would write applications onto punched cards and run them on large computer systems before it printed out the result for you. This was called running a job. Today the punched cards are no longer used, but the process of job submission is fairly similar. A job is a set of one or more executables to be run with a set of one or more arguments. The job eventually returns a result that the user can retrieve at a later point as demonstrated by figure 2.1.

```
Universe   = vanilla
Executable = /some/path/multiply
Arguments  = 5 5
Log        = multiply.log
Output     = multiply.$(Process).out
Error      = multiply.$(Process).error
Queue

Arguments = 5 10
Queue

Arguments = 15 15
```

```
Queue
```

Listing 2.1: Simple job descriptor for the HTCondor batch system that queues three jobs with different parameters.

2.3.1 Job Descriptor

Grid jobs are usually defined in job descriptor files that can be submitted to the grid, like the example in listing 2.1. There are several different job descriptor languages for various batch systems, and they can often be translated from one language to another. The job descriptor describes various parameters for the job, like the executable, the file for **stdin**, where to place **stdout** and **stderr** and various arguments for the job[5].

In the example in listing 2.1 the **/some/path/multiply** executable is used. A log file for the execution part of the job is selected (**multiply.log**), which will include details of how, when and where the job was executed. **stdout** and **stderr** are placed into **multiply.$(Process).out** and **multiply.$(Process).error**. **$(Process)** is one of several available macros that will be changed to individual values for the job. The **$(Process)** macro will be changed to the process id of the job, and since the job descriptor expands into three different jobs, it will be **0**,**1**, or **2** depending on which job is executed. When submitting to **HTCondor** the **multiply** executable will be sent to the batch system, as well as the job descriptor, and be expanded to three jobs that execute **multiply** with different parameters.

2.3.2 Arguments

Often in the grid world one wishes to run an executable with many different sets of arguments. A simple example would be to have an executable that divides a task into N pieces. Most job description languages have ways of dealing with this called a **parameter sweep**. Parameter sweeps enables multiple sets of arguments for a single job file, which will create multiple jobs with the different sets of arguments. In the example above one could create a job file that contains every argument from 0 to N, and the executable could then execute all the pieces of the task in parallel, all executed from a single job file.

Some job description languages also allow more advanced functionality like argument ranges where one can define a range of arguments that should be processed.

2.4 Virtual Organization

Grid systems usually have several different projects with a common goal, and hierarcically these projects are organized into virtual organizations. A virtual organization is a group of individuals, often several hundred to thousands, who work on the same project, and share resources between each other. A resource in this context is usually an immense catalogue of data, for example physics data from an experiment at the Large Hadron Collider[6] (LHC), a large database of some type, or equipment used by the project. Resources can also be shared between multiple virtual organizations.

Virtual organizations can vary in size, scope and duration, and often cross other organizational boundaries. Virtual organizations allow resources like computational time to be distributed according to administrative policies and can divide a finite amount of computing resources between the various goals of the grid.

2.5 Public Key Infrastructure

Public key infrastructure (PKI) is a cryptographic scheme for secure communication and validation of information, including the identity of the sender. PKI uses asymmetric cryptography to authenticate a user and securely communicate. Grid security infrastructure (GSI)[7] relies heavily on public key infrastructure.

2.5.1 Asymmetric cryptography

Asymmetric cryptography uses a pair of keys, where one key is distributed widely, the public key, while the other is only kept by the user, the private key. Both keys can be used to encrypt messages, and the other key can be used to decrypt the message.

Data and messages that are to be sent secretly to a user is encrypted using the receivers public key. Only the intended recipient can then decrypt the data or message by using his private

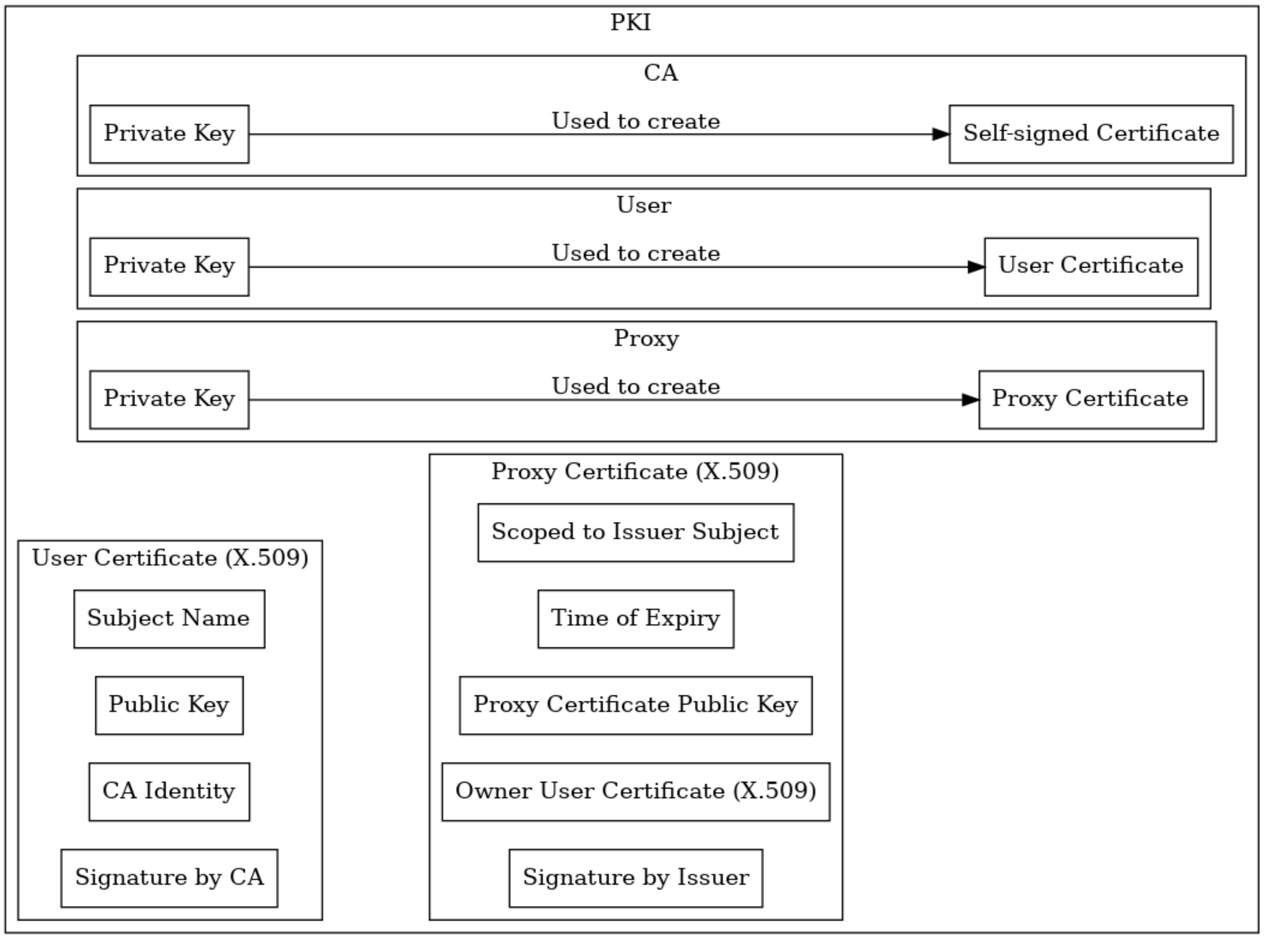

Figure 2.3: Different parts of the grid PKI.

key.

If the purpose is to validate that a user is who he says he is, one can send that user a random string to be encrypted, and if the encrypted message that is returned can be decrypted with that users public key, one can be certain that the user has the corresponding private key. The private key should be protected. This is done using filesystem read-protection and often a passphrase that is required to unlock the private key.

Asymmetric cryptography can fulfill the following security requirements:

Confidentiality Making sure only the intended recipient can read the message.

Integrity Making sure the message has not been altered.

Authentication Making sure a user or sender is who he says he is.

Non-repudiation Making sure the user can't deny that he sent a message.

A certificate is a file that connects a keypair to an identity. The certificate contains a user's public key, the user's name (subject name), the identity of an authority that has verified that the person who owns the public key is infact the user, and a cryptographic signature of the rest of the file by the authority. A requirement for being able to trust the user's certificate is that all parties trust this authority, which is called a Certificate Authority (CA).

2.5.2 Certificate Authority

The CA has a certificate and private key as shown on figure 2.3. All parties within the GSI, or the grid in this case, have the CA's public key. The CA's responsibility is validating the identity of users and connecting a user identity with a certificate. One way to do the validation is for the user to physically meet with the CA and authenticate himself.

A certificate request is created. The certificate request contains the subject (user) name, the subjects public key and the CA's identity. The certificate request is then signed by the CA to create the user certificate. When someone receives the certificate they can validate that it is signed by a CA they trust, and by extension they should also trust it. The CA's certificate is signed by the CA itself, and is called a self-signed certificate.

2.5.3 CA in Grid

A grid is constructed from computers in many different administrative domains. It is imperative that the workers have a way to trust the users submitting jobs are allowed to do so and are not authenticated independently in each administrative domain. This is done in the GSI using the CA. In a grid setting all participants implicitly trust the CA and the CA's validation process of users. The CA's validation usually means the user either meets directly with the CA and authenticates himself, or meets with someone the CA trusts[1] and authenticates himself to that person. The registration authority can for example be a users office or an administrator of that user whom is trusted by the CA.

Jobs can run for several hours or even days, and users or the users job needs to be able to authenticate to workers or services that require authentication, but leaving the private key unprotected would be a security risk. The user can solve this by delegating privileges by using proxy certificates.

2.5.4 Proxy Certificate

Services often need to communicate with other services in the grid on behalf of the user. An example would be if a worker node needs to upload the result of a calculation to a GridFTP-service. GSI implements this using a **Proxy Certificate**, which is an extension of the **Certificate standard**[8].

A proxy certificate contains a subject, which is limited to the scope of the issuer. In GSI a proxy certificate contains a time of expiry, although this is optional in the **Proxy Certificate** standard. It can optionally also contain delegation policies that decide *which* rights the issuer wishes to delegate.

A public key is used for the proxy certificate. The proxy certificate also contains the issuer's full certificate as well as a signature by the issuer, and an extension to identify it as a proxy certificate[8].

The issuer is either a user, using a normal user certificate, or another proxy using a proxy certificate. In both cases there is a verifiable path back to the issuer and the CA, including all

[1]This is called a Registration Authority (RA) in PKI

certificates in the path, so all limitations set by the issuer or proxies along the way can be verified by the receiver of the certificate to only give the permissions the issuer has given.

The private key matching a proxy certificates does not use a passphrase, so the issuer doesn't have to input the passphrase every time.

2.5.5 Security

A grid has a lot of resources, and in most cases it must be ensured that the user submitting a job to the grid is authorized to do so, and not a malicious actor. This is usually done using the GSI and **user certificates**. The CA is an integral part of the GSI, and is a central component of each grid. There is usually also a dedicated grid proxy service that enables the user to create and distribute a proxy certificate of his user certificate.

The proxy certificates are usually kept on a proxy service within the grid infrastructure, and have low lifecycles, often being valid no longer than 24 hours. Proxy certificates are used by grid middlewares in order for them to act on behalf of the user to interact with the various other services in the grid.

Various services that needs to be performed on behalf of the user are done using a proxy certificate the user has issued, which can be validated by all receiving parties.

Chapter 3

Cloud computing

Cloud computing is a model for on-demand access to a shared pool of computing resources that can be allocated and released by the consumer. According to *The NIST definition of Cloud Computing* by Peter Mell and Timothy Grance [9] a cloud has five essential characteristics.

> "*On-demand self-service.* A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed automatically without requiring human interaction with each service provider.
>
> *Broad network access.* Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, tablets, laptops, and workstations).
>
> *Resource pooling.* The provider's computing resources are pooled to serve multiple consumers using a multi-tenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. There is a sense of location independence in that the customer generally has no control or knowledge over the exact location of the provided resources but may be able to specify location at a higher level of abstraction (e.g., country, state, or datacenter). Examples of resources include storage, processing, memory, and network bandwidth.
>
> *Rapid elasticity.* Capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

> *Measured service.* Cloud systems automatically control and optimize resource use by leveraging a metering capability at some level of abstraction appropriate to the type of service (e.g., storage, processing, bandwidth, and active user accounts). Resource usage can be monitored, controlled, and reported, providing transparency for both the provider and consumer of the utilized service."

There are also several types of cloud services which *The NIST definition of Cloud Computing*[9] names deployment models. They define them as the following.

> "*Private cloud.* The cloud infrastructure is provisioned for exclusive use by a single organization comprising multiple consumers (e.g., business units). It may be owned, managed, and operated by the organization, a third party, or some combination of them, and it may exist on or off premises.
>
> *Community cloud.* The cloud infrastructure is provisioned for exclusive use by a specific community of consumers from organizations that have shared concerns (e.g., mission, security requirements, policy, and compliance considerations). It may be owned, managed, and operated by one or more of the organizations in the community, a third party, or some combination of them, and it may exist on or off premises.
>
> *Public cloud.* The cloud infrastructure is provisioned for open use by the general public. It may be owned, managed, and operated by a business, academic, or government organization, or some combination of them. It exists on the premises of the cloud provider.
>
> *Hybrid cloud.* The cloud infrastructure is a composition of two or more distinct cloud infrastructures (private, community, or public) that remain unique entities, but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load balancing between clouds)."

There are several potential advantages to using a cloud system, rather than statically provisioning hardware for tasks. If the consumer is using a public cloud the consumer will only pay for the resources being used, and not the cost of running the entire cloud and share the cost of the infrastructure with the other consumers, as opposed to having dedicated infrastructure to your project, which would bring the cost up.

All cloud types, both private and community clouds as well as public clouds, also enable re-purposing of resources on a day to day, and even hour to hour basis. A project which may require a lot of resources at one point may not require them all the time, which allows re-purposing of those resources to other project when not in use and potentially increasing resource utilization. For a larger project with multiple subprojects that may have different computational needs and goals, building a private or community cloud makes a lot of sense as you can centralize the management of the cloud without the need for each subproject to have their own hardware, and utilize the available hardware better.

There are additional advantages of using a cloud, like the On-demand self-service, which enables consumers and applications to provision and release computing resources through APIs, rather than requiring access to the hardware. Using a cloud solution one can write an application that provisions a server when the load requires it and releases it when it is no longer required.

3.1 Cloud solutions

There are many different platforms for implementing a cloud including Nimbus, OpenNebula and OpenStack. Nimbus and OpenNebula used to be popular, but in the more recent years most users are migrating over to OpenStack. The *ALICE Tier1/Tier2 workshop* in 2016[10] confirmed this as all the ALICE users have either migrated or are in the process of migrating to OpenStack in the near future. For these reasons this paper focuses more on OpenStack than alternatives.

3.2 OpenStack

The OpenStack project was launched in 2010 by Rackspace Hosting and NASA. OpenStack is a free and open source infrastructure-as-a-service (IaaS) cloud computing software platform that originally started out with code from NASA's Nebula platform and Rackspace's Cloud files platform. It has since grown massively with over 500 companies joining the project[11]. OpenStack aims to provide a complete solution for all parts of a cloud infrastructure, and be customizable enough to fit all the needs a cloud provider could have. OpenStack is also modular enough that all components can be replaced, either by other components developed by the OpenStack teams that perform similar tasks but may have different ways to be configured, or by third party modules that can replace the component in question. An example would be the networking component - for this Nova can be used, as it has build in, but simple network handling logic, or one can use Neutron which is a highly complex, highly configurable networking solution that can create complex hierarchical virtual networks within the virtual data center.

OpenStack fulfills the five essential characteristics of a cloud in the following ways:

On-demand self-service OpenStacks account service, **Keystone**, implements the concept of projects, or tenants which is synonymous in this context, which is a collection of users cooperating on towards a single goal. A project can have multiple users, and many sets of restrictions put on it, like how many cpu cores it can provision for VMs or how much disk space it may use.

A user can provision resources on-demand within the limits set by the project either through the web interface **Horizon**, the **OpenStack commandline interface**, or the **REST api**.

Broad network access Access to OpenStacks features is implemented by several APIs including the industry standard EC2 API. In addition to EC2 OpenStack implements a standardized API for its features through the REST protocol. There are also command-line clients and language bindings for python available as abstraction layers ontop of the APIs.

As the REST APIs use the http protocol they are available from just about every platform in use today and can be used on everything from large servers using the APIs in an automated fashion, to small Internet of Things (IoT) devices that could use OpenStack to perform some tasks that requires more power that the device has available.

Resource pooling OpenStack allows setting multiple limits on tenants, including on cpu cores in use, memory constraints and disk usage. The limits on resources for all tenants do not need to add up to less than the resources available. This allows over-provisioning of resources which leads to sharing resources at times when not all resources are in use.

For example if there is 1 TiB of disk space available and you limit 3 users to 500 GiB each they can all share the available disk space as long as they in total provision less than what is available.

The tenants can not see how much of each resource is available - only how much they are limited to.

Rapid elasticity OpenStack has two modules that provide rapid elasticity - **Senlin** and **Heat**. Heat is a module that aims to provide automated orchestration of VMs deployed in a cloud. Heat will be mentioned a lot more in 3.5.1. Senlin is a module that aims to create a generic clustering service for OpenStack clouds. Senlin builds on top of Heat (Orchestration), Nova (VM services) and Cinder (Volume Management) and aims to provide more flexibility than what Heat can provide with its clustering capabilities. Heat will be described more in depth in 3.5.1.

Both Heat and Senlin allows the user to create a cloud which is elastic by scaling dynamically based on resource usage up to the limits placed on the tenant.

Measured service OpenStack provides detailed metrics on resource usage with the **Ceilometer** module which gathers detailed samples of resource usage and can provide detailed statistics on how much of any resource a tenant has used.

Ceilometer will be described more in depth in 3.6.

3.3 OpenStack components

Nova

Nova[12] is a service that manages virtual machines. It is a large project with several sub-projects, including Nova-compute which runs on all the VM hosts in the cloud, and Nova-Scheduler, which coordinates which VMs should on each compute node and controls the compute nodes. The goal of Nova is to provide self service access to compute resources, and Nova functions as an umbrella project for the sub-projects that provide this.

Cinder

Cinder[13] is a Block Storage service for OpenStack. It is a service that provides disk space to VMs in an OpenStack cloud. The user rarely needs to communicate directly with Cinder, as the other OpenStack services often do this when needed.

Neutron

Neutron[14] is a service that provides software defined networking for VMs in an OpenStack cloud. It supports implementing complex networking architectures in a virtual environment. Neutron provides networking interfaces, virtual switches and routing between interfaces, regardless of the underlying technology providing the physical routing between machines.

Neutron was created to enable OpenStack to fulfill advanced requirements that were not handled by the older networking service Nova-network, and is today the default networking provider for OpenStack.

3.4 Senlin

Senlin[15] is a generic clustering service for OpenStack that builds on top of the features exposed by Heat, which will be talked more about in 3.5.1. The project started in late 2014/early 2015[16] and is under active development. A stack based on Senlin provides more control and management than a stack based on Heat. As an example Heat has no mechanism for selecting *which* VM should be deleted when scaling down, so it selects the oldest one. Senlin allows the user to add and destroy *specific* VMs at will.

Senlin also has more scaling options than Heat and can scale the number of nodes up and down by a percentage, a static number or statically set the size to a number; where as Heat is only able to scale up or down by a static number.

Senlin also allows the user to update values for the stack (for example which image new VMs should use) with a single command using the Senlin client, while Heat requires that the user updates the template used to create the stack and then passes the updated template to the stack using **heat stack-update**.

The downside is that Senlin is a relatively new project, and the documentation is lacking, or non-existent for a lot of features. It is not much in use yet, and therefore real-world working examples are hard to find. Once the project is more mature and has more documentation it will likely be a powerful tool to create and control an OpenStack cluster, but as of yet I have chosen Heat for this project as it is more mature and has complete documentation.

3.5 Heat

3.5.1 What is Heat?

Heat[17] is the module in OpenStack that performs orchestration.

> Orchestration is the automated arrangement, coordination, and management of computer systems, middleware, and services.[18]

The above is an unsourced quote from Wikipedia, but it fits very well with what services Heat perform. Heat can arrange VMs that interact with each other, coordinate services within those VMs to talk with each other, and manage services on the VMs it creates. Heat provides orchestration of OpenStack resources using modular templates.

Heat uses templates in the Heat Orchestration Template (HOT) format which uses the yaml data format syntax. The template format implements various other OpenStack modules as resources which can be stringed together in the HOT template and used by the Heat engine, which then communicates with those other services in order to perform the orchestration task. A resource in this context can be, among many other things, a disk image, a network interface, or an alarm to trigger some action.

As figure 3.1 shows the user communicates, using the heat or openstack clients, with the Heat engine, which then communicates with Nova for VM management, Neutron for network management, Cinder for disk and VM-image management, and Ceilometer[1] for monitoring, statistics and data about the VMs. The optional Heat-api-cfn service enables callbacks for various parts of the Heat engine, which enables services like Ceilometer to communicate back with Heat.

[1]Ceilometer will be mentioned a lot more in 3.6.

This is also the way alarms work in Heat. Heat will use Ceilometer to create an alarm that Ceilometer will check for, and include a callback url to Heat-api-cfn in the alarm. When the alarm triggers within Ceilometer it will open the callback url which makes the Heat engine perform the action the HOT file defines for that alarm.

The templates for Heat have three sections:

The heat_template_version: This variable is required. **2016-04-08** is the latest version at the time of writing, and as such will be used here. This version number defines which features are available in the template format. A list of the features available in each version, including the upcoming *Newton* version exists here: `http://docs.openstack.org/developer/heat/template_guide/hot_spec.html#hot-spec-template-version`

parameters: All the parameters go in this optional section. Things that the stack creator may want to redefine, like usernames and passwords, should be in this section. In the templates where the values defined here need to be used, the **get_param** intrinsic function which looks up the value from the parameters and replaces the parameter lookup with the parameter value.

resources: This required section is the core of the stack where all the resources for the stack is defined. All the virtual machines, the VM properties, alarms and other resources are defined in this section. Resources can also have dependencies on other resources - for example a Wordpress webservice VM instance resource can have a dependency of the database service on the database instance being running, which would prevent starting the Wordpress instance until the database instance is fully started and has initialized the database.[20]

outputs: This is an optional section where any information to be propagated out of the stack is defined. Passwords can be randomly generated, but if the user needs a way to get access to the passwords they can be defined in the outputs section as an output. Any other information, like VM count, and other properties that the user should know about could also be defined here. The outputs can be listed using the **heat output-list** command, and showed using the **heat output-show** command.

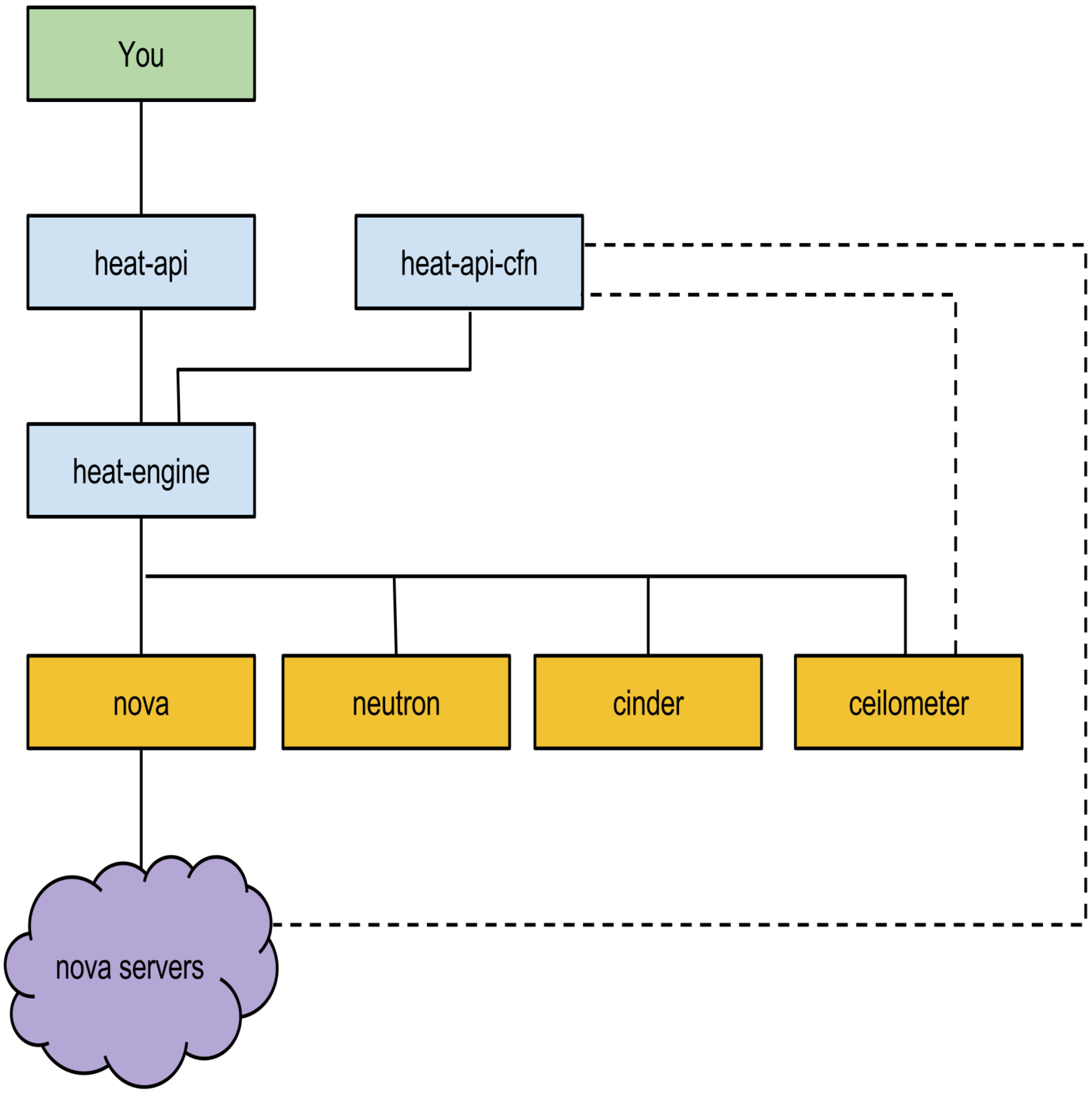

Figure 3.1: Heat architecture overview figure from [19]

It is possible and highly recommended to split up the definitions into multiple files as it would be unmaintainable to have everything in a single large file. For this another HOT file, called the environment file, is used. The environment contains the **resource_registry** which is a lookup table for external resources. The **resource_registry** resource contains key/value pairs where the key is the resource name, and the value is the URI to find that resource. A common use for the environment is to define VM templates in separate files and just define *which* VM template to use in the main HOT file.

Resources can also be referred to directly by their filename as the resource *type*, which will include that resource in the current resource. Any resource names that end in **.yaml** will be included. This is a distinction from using the resource registry, as the registry requires a full URI that is accessible from the **heat-engine** service and will be included when the resources are being created, while using type will be included by the client used to submit the query to the **heat-api**.

Heat is a powerful tool, but it has no sensors for measuring things like system load or resource usage, which is a requirement for being able to call OpenStack a *cloud*. Instead Heat can use the OpenStack Telemetry services for this.

3.6 Telemetry

The Telemetry services are the components in OpenStack that are responsible for metering various resources like cpu, memory and bandwidth usage of VMs, logging them to databases and calculating statistics on these. OpenStack has three subprojects that make up its telemetry services - **Aodh**, **Ceilometer** and **Gnocchi**. The services perform various services that together make up a complete platform for metering, and measuring various statistics and performing actions when those statistics go outside of thresholds.

Ceilometer Ceilometer[21] is a service that collects samples of the physical and virtual resources comprising an OpenStack Cloud and stores them in a database for later processing. It can run in two modes - as an agent on the system in question, or passively by polling the hypervisor for information. Polling is more resource intensive, but is more generic and requires less intervention by not requiring the installation of an agent onto the cloud image.

It is also possible to make custom agents and in other ways send samples to Ceilometer, but this is outside of the scope of this thesis.

Gnocchi Gnocchi[22] aims to provide several features, many of which can also provided by Ceilometer, but without the scaling issues Ceilometer can face when storing large amounts of data. Where Ceilometer needs to go through all the samples it has stored each time requests for certain metrics come in[23], Gnocchi stores samples in a time-series database, which can perform most requests in O(1) time as opposed to Ceilometer where most requests take O(n) time.

Aodh Aodh's[24] goal is to provide alarms and notification based on the metrics it receives from Ceilometer and Gnocchi. An alarm can be set on a single meter, or a combination of meters. Aodh allows the user to set alarms based on evaluation of a collection of samples from Gnocchi and Ceilometer or a dedicated event. Currently when an alarms goes off Aodh can perform three distinct actions:

- HTTP Callback - Aodh can call a URL to notify a service that the alarm has been triggered. Heat creates a callback url to the Heat-api-cfn when it creates an alarm and uses this method to be notified when that alarm is triggered.
- Log - Aodh can log when an alarm is triggered.
- Notification message. Aodh can send a notification upon an alarm triggering to a messaging service through the Zaqar API.

Aodh does not have an interface of its own and to interact with it the Ceilometer interfaces are used.

The OpenStack Telemetry services originally started out as just Ceilometer which was used to provide a customer billing and resource tracking service. The project grew in the amount of meters it supported and ways to normalize and transform data from all the OpenStack core components, and the project was placed under the Telemetry project umbrella, which include the other projects that interact with Ceilometer.

An example use case for these three services is a user that wants to start more VMs when cpu usage within a timeframe exceeds 50%. The user sets an alarm in Aodh to be triggered when the cpu usage within the timeframe exceeds 50%. Ceilometer gathers samples from the machines in question; and Aodh can query Ceilometer or Gnocchi for the metrics across the time frame.

This is exactly what we will do with Heat. When creating a new stack with an **AutoScalingGroup** Heat will create alarms in Aodh for each condition where scaling would occur will a callback url to the Heat-cfn-api which makes sure Aodh can communicate back to Heat. Aodh will then update the status of the alarms at a defined time interval, and notify Heat when the status changes.

Chapter 4

Virtual Machines

Virtual Machine is a term that has existed for a long time, and has a slightly different meaning depending on the concept it refers to. In this thesis the term refers to system virtual machines of the bare metal, or hosted types.

A virtual machine (VM) is a computer implemented in software that, like a physical computer, runs applications and an operating system. The VM runs just like a normal computer with a processor and memory, but several VMs can run at the same time, sharing the physical hardware.

Some instructions on the virtual machine are translated to other instructions before being run on the physical hardware. The application that performs this operation is a hypervisor (also known as a **Virtual Machine Manager**) using hardware-assisted virtualization. This gives a VM interesting security properties, including isolation, as the memory space from a VM can be translated into an isolated memory space on the physical hardware. Utilizing virtualization, binary code can run identically regardless of what the host operating system or hypervisor is.

4.1 Introduction

Virtual machines are software platforms that provide the means to run virtual appliances or entire operating systems on virtual hardware. They enable the user to run multiple operating systems in parallel on a single set of hardware, without the virtual machines being able to interfere with each other. There are two main types of virtual machines - type-1 or bare-metal

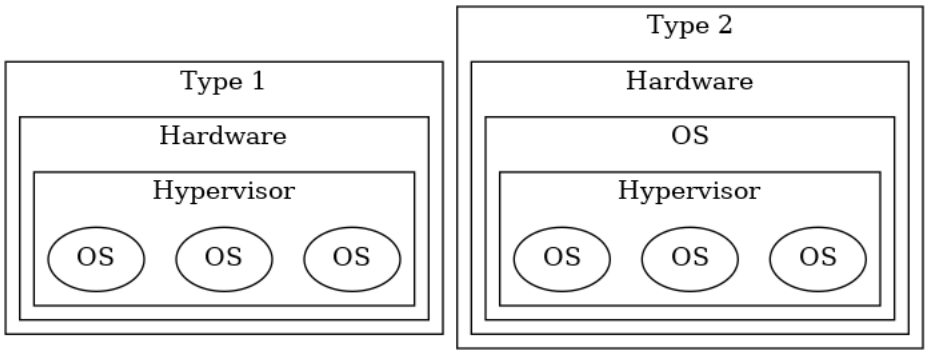

Figure 4.1: Hypervisor types.

hypervisors (also known as native hypervisors) and type-2 or hosted hypervisors. Figure 4.1 demonstrates the difference between these.

Hosted hypervisors run on a host operating system, but require a driver in the host operating system's kernel. This driver performs the hypervisor-specific tasks, and the virtualization application communicates with this driver to perform the virtualization functions and enable guest VMs to run. Examples of hosted hypervisors are VMware Workstation, Virtualbox and QEMU.

Type-1 or bare-metal hypervisors run directly on the hardware and function as a thin abstraction layer between the hardware and the virtual machines. They manage guest operating systems, often through a special privileged guest VM that gets special permissions to control the hypervisor. Examples of bare-metal hypervisors are Xen, Microsoft Hyper-V and VMware ESX/ESXi.

There are also a class of hypervisor that fits somewhere in the middle of type-1 and type-2. Linux's Kernel-based Virtual Machine (KVM) and FreeBSD's bhyve are kernel modules that turn the host operating system into a virtual machine hypervisor. This in effect turns the kernel into a type-1 hypervisor that runs directly on the hardware, but with an entire operating system competing with the hypervisor for hardware resources. Because of sharing resources with an operating system and requiring an operating system in order to run they can also be categorized as type-2 hypervisors.

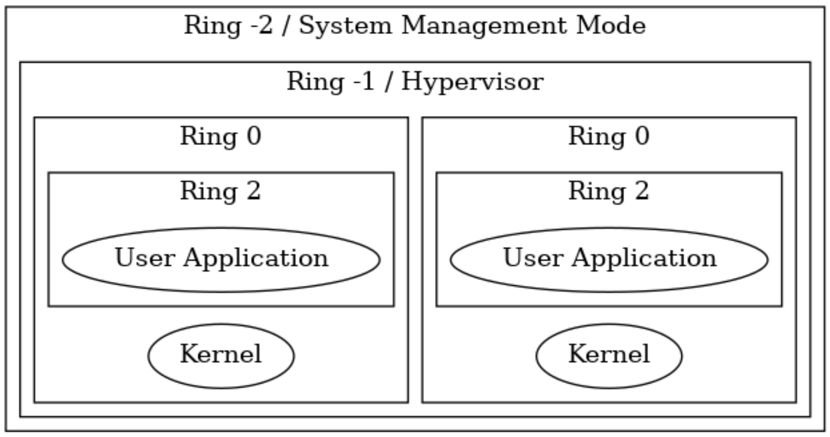

Figure 4.2: Privilege rings.

4.2 Security

4.2.1 Isolation through protection rings

Virtualization allows several VMs to be running on the same system, and it is important that they are isolated from each other so a malicious virtual appliance could not look at the data from other virtual machines or take over the system. Isolation is implemented through the use of protection rings.

As figure 4.2 demonstrates computers today use several protection rings which decide which instructions an application can and can not run. In most cases operating systems only use 2 of the available protection rings - ring 3 (or *usermode*) for user applications, and ring 0 (or *kernelmode*) for the kernel and device drivers. Protection rings are also used to limit which regions of memory an application has access to, as the instructions to set the memory limits of an application require ring-0 access.

Earlier virtualization solutions implemented VM isolation by moving the virtual appliance's kernel from ring 0 to the unused ring 1, and used interrupts to take over when the appliance tried to do something that requires ring 0 access. As some x86 instructions require ring 0 access[25] those instructions either had to be emulated (which is slow) or the virtual appliance's kernel had to be modified to avoid requiring those instructions.

Today CPU manufacturers solve this another way, by implementing an additional protection ring - **ring -1**. By having the hypervisor run in ring -1, and isolate virtual machines from there, virtual appliances can run in ring 0 unmodified. The isolation is implemented by having certain instructions cause an interrupt to ring -1 and having the hypervisor running there execute or rewrite the instruction before execution. Memory is also mapped to different regions on the physical memory, ensuring a VM can not read the memory content of another VM.

There is also another protection ring - **ring -2** which manages features like power management, the cpu killswitch code that triggers if the cpu gets too hot and the Trusted Platform Module (TPM) and can ensure no malware can boot into ring -1 and take over the system instead of the hypervisor. These are not included in the model in figure 4.2 as they are an extension to the traditional privilege ring model. These extensions are implemented by having certain interrupts, or in ring-2's case memory ranges, go to ring-1 or ring-2 instead of directly to the hardware.

4.2.2 Lifetime

Almost all the hardware in VMs is abstracted away, which enables the ability to quickly create and remove VMs as well as move running VMs between different physical computers. Since VMs are fast to create and remove it is feasible to create a new virtual machine to execute a task and then delete it as soon as the task is done. Security is slightly improved by this, as malicious software that was to infect the VM will only stay until the VM is deleted, at which point the malicious software itself will also be deleted.

QubesOS[26] is an operating system that uses virtualization for security and can start a new virtual machine every time the user starts the web-browser. Since the new VM is deleted when the user closes the web browser, this effectively protects against all types of browser exploits.

Using VMs can also increase uptime, since if a physical computer needs maintenance all virtual appliances running on that computer can be moved to another computer until the maintenance is done. When maintenance is done they can be moved back, allowing VMs to achieve higher uptime than the hardware they run on.

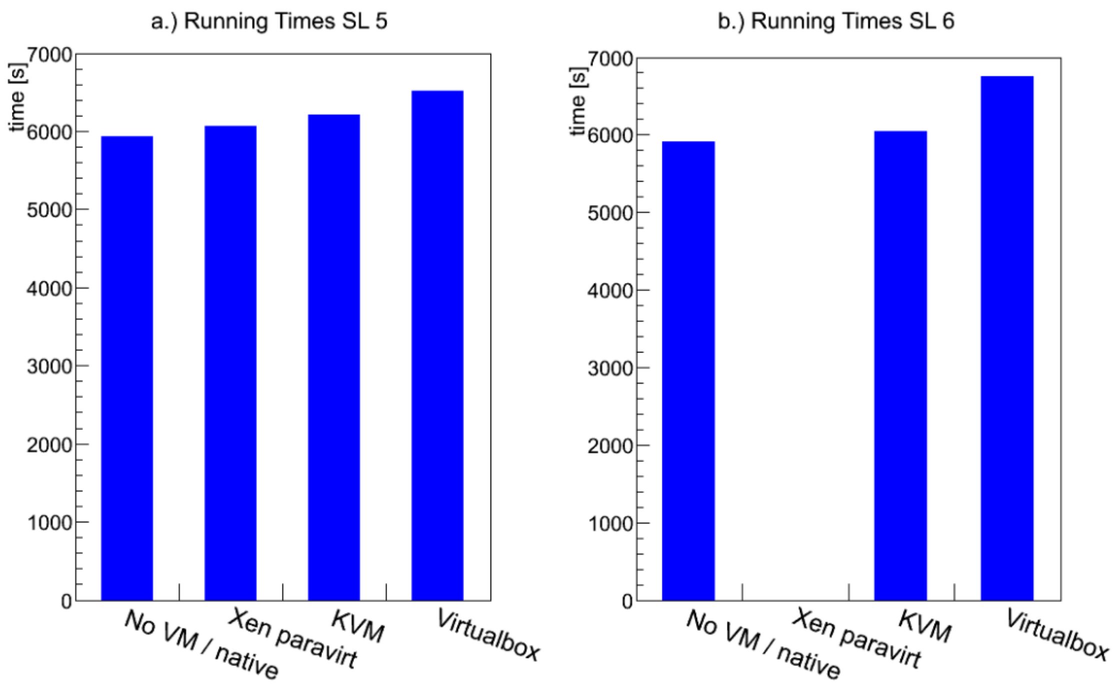

Figure 4.3: Execution time of ALICE benchmark test program on various virtualization technologies. Figure from [27].

4.3 Performance

Because of hardware support for virtualization, almost all instructions are executed directly on the processor. Because of this the performance of VMs is about the same as a normal physical computer. There is some overhead for the hypervisor on a few instructions, but as the hypervisor is just a thin layer between the VM and the hardware, the speed is close to native. Figure 4.3 from *Dynamic virtualization tools for running ALICE grid jobs on the Nordic ARC grid*[27] is a performance comparison of a benchmark ran under various virtualization technologies.

4.4 Virtual Appliances

There are several virtual appliances built for running on virtualization platforms. Usually these are created as disk images that can be booted directly from with no need for an installation. One such virtual appliance is CernVM.

4.4.1 CernVM

CernVM[28] is a virtual appliance created to be used by the participants of the CERN LHC experiments. It's a virtual appliance that is extensible, portable and easy to configure for developing and running LHC physics software locally, in a grid or on a cloud provider.

The image consists of a modified version on Scientific Linux[29], a small Linux kernel and the CernVM-FS application. CernVM-FS is an application that allows files to be downloaded on demand, rather than being stored locally. Since CernVM 3, the virtual appliance is distributed using the µCernVM bootloader with a read-only image of only about 20 MiB, with most of the files being streamed from the internet. This makes it ideal for use in a cloud as the provisioning of new machines is fast due to the small disk image, and little software needs to be installed, as the image already comes with a lot of software ready to be streamed from the internet.

Chapter 5

Other Grid/Cloud Interfaces

There are mainly three other projects exploring how to leverage cloud in the grid, and have different approaches on how to best do this. These are Elastiq, AliEC2 and an implementation that was presented at the HEPiX Fall 2013 workshop.

5.1 Elastiq

5.1.1 Introduction

Elastiq[30] is a daemon that enables a cluster of virtual machines to scale up and down automatically. It is built using Python, so it should be able to run everywhere and uses the EC2 api to communicate with the cluster, so it should be able to use both OpenStack and Amazon's cloud to provide virtual machines.

In order to do scaling Elastiq monitors two different metrics. To scale up Elastiq monitors the batch system's queue of jobs, and if there are too many jobs waiting it requests new virtual machines through the EC2 api. The batch system is polled every 15 seconds, and if there are more than 4 jobs in the queue per VM that is running, it increases the number of VMs to what is needed to get below 4 jobs running per VM.

The part of Elastiq that scales down the VMs checks how long VMs have been idle for every 45 seconds and removes VMs that have been idle for an hour or more. This ensures only VMs that aren't working on any jobs get removed and keeps the number of VMs in use around the minimum required, without scaling down too fast, which could create a wave-like effect in the amount of VMs running.

5.1.2 About the project

Elastiq interfaces directly with HTCondor, and as such has a hard dependency on using this batch system. It is possible to write plugins for other batch system, but this isn't done yet.

The project has a low amount of activity and as of the time of writing the last commit to the project is from 15th of June 2015 (over 1 year ago).

The system relies heavily on EC2 and the batch system use case, and it would not make sense to attempt to fit other use cases into the scope of Elastiq. The prototype in this thesis takes a more generic approach and is more suited to be adapted to other use cases than the one of batch systems.

5.2 AliEC2

5.2.1 Introduction

AliEC2 [31][32] is a project to use virtual machines to execute jobs from AliEn. It is implemented as a grid job scheduler for the AliEn grid project. The project uses the EC2 API to provision VMs on an EC2 compatible cloud solution like Amazons cloud or OpenStack, and does some basic monitoring to release idle, or dead VMs.

5.2.2 About the project

AliEC2 interfaces directly with AliEn by replacing the scheduler component of AliEn. Since AliEC2 is so tightly coupled with AliEn it is unsuited for usage as a scaling layer for non-AliEn applications.

5.3 Rutherford Appleton Laboratory Tier-1 system

5.3.1 Introduction

Andrew Lahiff, Ian Collier and Orlin Alexandrov presented work at the HEPiX Fall 2013 workshop on a system[33] that uses virtual machines with HTCondor to create an elastic grid.

5.3.2 About the project

The project uses StratusLab, which is based on OpenNebula, as a cloud provider. Similar to this prototype they use CloudInit to instantiate VMs. During instantiation they insert the password for the HTCondor pool. They do not mention how networking is done in this project. In the prototype in this thesis HTCondor is run in a closed system, and as such authentication is disabled as it is not required.

Another difference is that in the RAL Tier-1 system condor is responsible for starting and stopping VMs, while in this prototype OpenStack's Heat is used instead. There are advantages to both methods - mainly:

Using HTCondor to detect if a VM is idle can give a more accurate result than using Telemetry. HTCondor knows when a VM is working on a job, even if it is using no resources. Telemetry relies on statistics about resource usage to determine if the system is in use or not. Because of this a job that is just sleeping could make a Telemetry-based system scale down because it detects very low resource usage, while a system relying on HTCondor would detect the worker as busy and may even scale up.

Using Telemetry Using a solution based on Telemetry is a more general-purpose solution and can be more easily suited to new use cases. A system using Telemetry can in a generic manner detect wether a resource is in use or not, while a system not using Telemetry needs to reimplement this logic.

At the time of writing the project is closed source and considered propriatary as no public repository for this project exists and there are no sources for how it is actually implemented. OpenNebula is outdated and most of those in the high energy physics domain that haven't phased it out already are in the process of doing so[10]. It is possible the team has updated the solution in use to OpenStack instead of OpenNebula, but no sources were found for the current status of the project.

Chapter 6

Solution

The prototype is an implementation of an elastic computational job environment that is able to execute jobs, and will scale up and down based on the load on the system. The actual implementation is done in HOT files, which are then used to make Heat create and destroy instances of the prototype environment.

This chapter gives a high-level overview of the design of the prototype. The prototype aims to prove how viable such a solution would be to use in a larger-scale, and uncover potential issues. The issues uncovered are documented in 7.5.

6.1 Introduction

The solution was designed with reusability in mind. For this reason the solution is separated into two independent sets of components - the internal and the external. The internal parts handle jobs execution and submission, while the external components handle measurements and scaling. Since they are separated the components can be reused or modified independently, for example using the external components to scale a different application. In the development of the prototype three machines were used as this should give some confidence the system will work with larger numbers.

6.2 Design Decisions

This section covers a few of the design decisions that have been made.

6.2.1 Cloud Infrastructure - OpenStack

The service for the cloud infrastructure was chosen to be OpenStack as it is the solution that is either in use in most places or going to be soon. The reason is outlined more thoroughly in 3.1.

6.2.2 OpenStack Installers

OpenStack is highly modular, and built to be able to fulfill a lot of different requirements for a Cloud Infrastructure. This makes it extremely customizable. Instead of installing and configuring everything manually there are several different installers to chose from that will perform the installation and configuration for you. **Devstack**[34], **ANVIL**[35], **Fuel**[36] and a set of **Puppet scripts** by Puppetlabs[37] were the installation methods found during this thesis. The Puppet scripts by Puppetlabs were attempted, but due to a lack of maintenance there were several issues encountered, and eventually the version of OpenStack they attempted installing was marked as End-of-Life.

Devstack and ANVIL are solutions for installing single-node development systems of OpenStack, and are unable to install OpenStack across multiple nodes. As a multi-node setup is used in this project these installers can not be used.

Fuel is a commercial product by Mirantis for deploying OpenStack clouds. It requires a dedicated machine to act as an admin node, and one paper describes the installation difficulty as high[38]. For the scope of this project, the added complexity of Fuel and requirement of an additional dedicated machine is unwanted.

Packstack is an opensource product that uses deploys OpenStack on multiple servers using SSH and Puppet. It is actively maintained, and the Puppet modules in use by Packstack are not the same ones used by the Puppetlabs module. It is the easiest to use solution that is able to deploy to multiple nodes.

6.2.3 OpenStack Installer - Packstack

Packstack is used for the installation. A few different methods of installation were attempted before settling on Packstack, but of the methods, Packstack was by far the easiest to use. The various attempts and installation is documented in chapter B, appendix A, and chapter 7.1.

It should however be noted that Packstack deploys an OpenStack environment that should work for the purpose of this prototype there are some stability concerns, and for a production environment the configuration made by Packstack may not be stable enough[38].

6.2.4 Virtual Machine Image - CernVM4

Currently most of the participants of the CERN LHC experiments use the CernVM image. This is an image which is heavily based on CernVM-FS, which is a filesystem driver that allows the machine to stream the filesystem from the CERN servers, rather than having it stored locally. Most relevant software is on the CERN servers and is in this way ready to be streamed on demand to the local image without needing to install it.

CernVM4 was used for this project. CernVM3 will not work, as there are some issues that are described in more detail in 7.5.8.

6.2.5 Batch System

There are several available batch systems to chose for running jobs. This prototype will have workers dynamically added and removed, so it is important that the batch system has a way to start using workers that get added without requiring a restart, and will detect when a worker is removed. HTCondor is the system that was chosen for this, as it has built in support for workers being added and removed, where other batch systems often require a service restart for this.

6.3 OpenStack Services

OpenStack has a fairly large overhead for the core services when the system is running on just one machine, but the overhead is mostly static and does not increase linearly when machines are added. For this reason all the core services run on a single machine, which will have very lit-

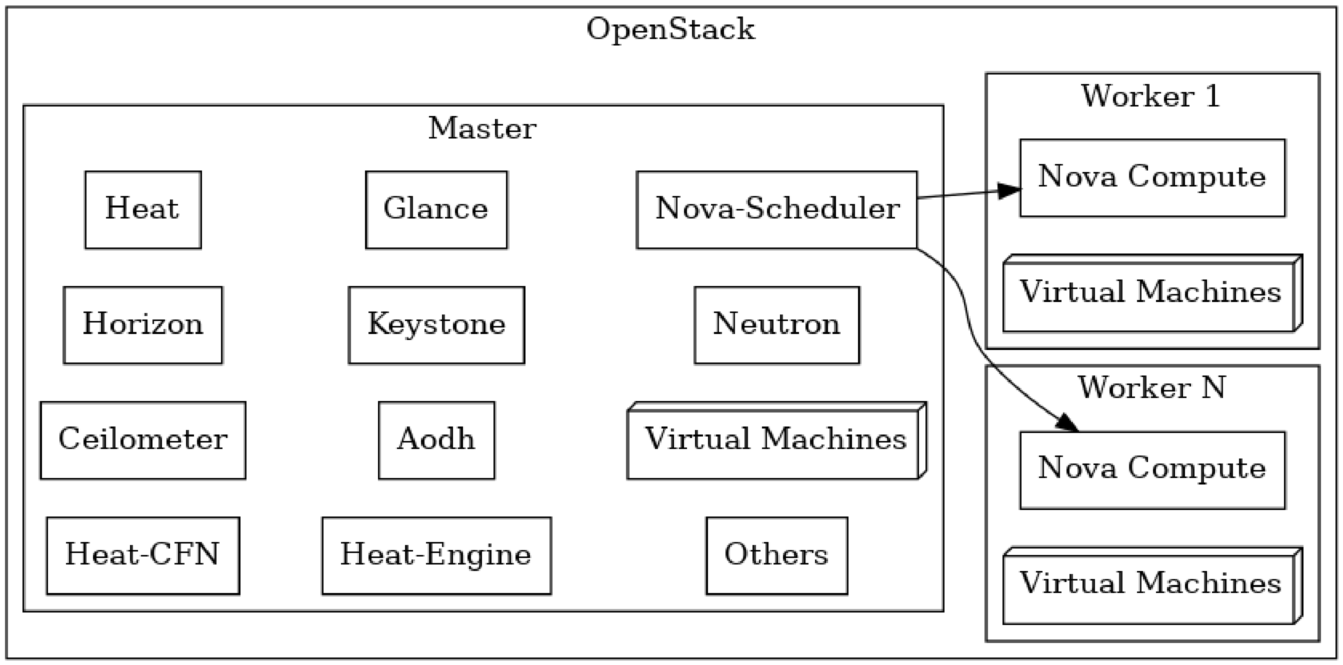

Figure 6.1: The services in the prototype.

tle resources left for running virtual machines, and two dedicated worker machines which only have the job of running virtual machines. During installation it was observed that the overhead of the core services did not increase significantly when compute nodes were added, so in a larger scale the overhead of the core services should not be a significant amount of the available computing resources. As an example, a CERN installation has scaled the controller components in a cell-like configuration to ensure the controller services are scaled to the amount of VMs they are servicing[39].

The OpenStack services communicate over REST APIs, so it does not matter much if the services are running on the same servers, or different ones. This makes OpenStack extremely modular, and enables increasing the number of Compute nodes easily. An overview of the services in use by the prototype can be seen in figure 6.1.

Nova

Nova is the service that manages virtual machines. Nova is divided into several services, but the ones in use by this project is only Nova-Scheduler and Nova-Compute.

Nova-Compute

Nova-Compute is the service that directly manages the virtual machines. Nova creates, destroys, suspends, snapshots and performs other actions related to the management of a VM.

Nova-Scheduler

Nova-Scheduler is a service that manages Nova-Compute services. Nova-Compute services connect to the Nova-Scheduler which will then add the resources from the machines running Nova-Compute to the pool of available resources.

As seen in figure 6.1 the central Nova-Scheduler controls the various Nova-Compute instances, which are the services in control of the virtual machines. Nova-Scheduler can automatically use any new Nova-Compute instances that are added to the installation without a need for adding them to a configuration or restarting any services.

Ceilometer

The Ceilometer service gathers metrics on VMs, like cpu-time used and memory usage, and organizes the data so Ceilometer can be queried for things like "average cpu utilization over past 2 hours". It can also do more advanced metrics, like averaging the cpu usage on a group of VMs instead of just one.

Aodh

Aodh is a service that can check some condition, and perform an action if the condition is triggered. It uses Ceilometer to check the conditions. An example condition would be "if cpu usage in past 2 hours was over 50%". The actions performed can be POSTing to a callback url.

Neutron

Neutron is the networking service, providing everything related to networking to virtual machines. Advanced networking and communication infrastructures can be implemented in Neutron.

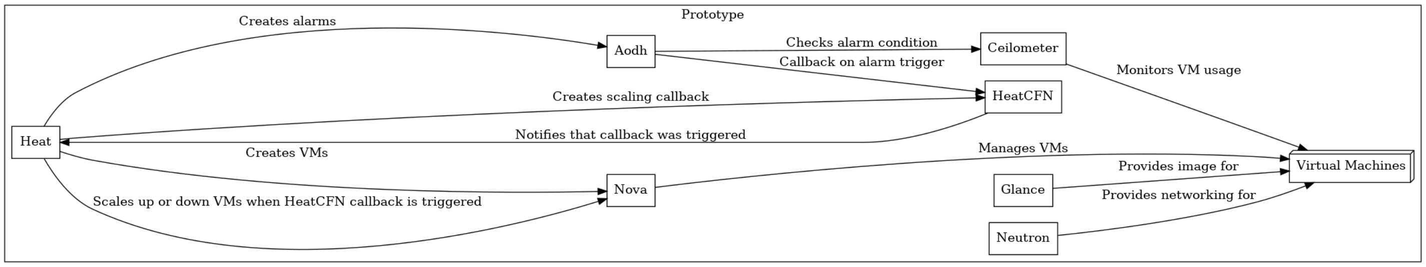

Figure 6.2: Overview of the prototype

Glance

Glace is a data store that stores and makes available VM images and virtual appliances. Glance is used to upload new images and manage existing images.

Heat-CFN

Heat-CFN provides callback URLs to functionality in the internal Heat-engine service. Using this one can define functionality in the Heat Orchestration Template that is exposed through the use of POST urls, and in this way can be triggered by other services, like Aodh.

Heat

Heat, which is described in more detail in chapter 3.5.1, is the service this prototype builds on top of. As demonstrated in figure 6.2 Heat is the service that interacts with other services in order for them to perform their respective tasks, without the need for the user to interact with them directly.

The prototype uses the Heat Orchestration Template(HOT) format to describe the interactions with the other services. The internal components of the prototype are defined in a HOT file.

6.4 Internal Prototype

The goal is to create a grid that can be scaled up and down dynamically. There are a few administrative tasks that needs to be done, like keeping track of the worker nodes in the grid and scheduling jobs for those. These tasks will be performed by a dedicated **master** VM (just named master from here on), which will not be executing jobs. Since the master will not be executing jobs it should not be part of the scaling calculation. For this reason the prototype was divided into two distinct VM groups - The master VM, and the worker VMs.

6.4.1 The Master VM

The master VM is a static VM that should handle all the administrative tasks required of the stack that isn't executing jobs. It should be the VM that receives jobs and hands them out to the running workers. It should also detect when new worker VMs are created and removed. The load on the master VM should not take part in the scaling of the prototype.

6.4.2 The Worker VMs

The worker VMs are the VMs executing jobs arrive. They need to detect the condor collector, running on the master VM, and connect to it in order to start receiving and executing jobs. The cpu load on the worker VMs should be measured, and if the average load exceeds a threshold, more worker VMs should be added. This threshold should be easily modifiable and is for the prototype set to 50% as an arbitrary value.

6.5 Batch System

The batch system must to be able to handle the dynamic use case of this solution, and HTCondors handles this type of dynamic worker allocation well. HTCondor is highly modular, consisting of modules that communicate with each other to form the complete batch system as shown in figure 6.3. The set of modules required to be running on a machine depends on the task that machine is intended for. The generic solution should work for other batch systems as well, with some changes depending on how the other batch system detects and adds workers.

6.5.1 Condor Master

The condor master is a process that starts, and watches over the other processes in a HTCondor pool. It is the only daemon required to be running on all nodes in the system. If any of the other condor daemons running on a node crashes or exits unexpectedly, the condor master will restart that daemon, and potentially notify an administrator. The condor master will also make sure to restart daemons if a daemon's binary is modified (for example by an update).

6.5.2 Condor Collector

Condor collector is the daemon in the pool that collects and stores information about all other condor daemons in the pool. All daemons send updates about their status periodically to the collector, and the collector can propagate this information on to other services that require it. This is the service that tracks how many worker nodes there are in the pool and the amount of computing resources available on each node. The condor collector also receives information from the condor scheduler about how many jobs that are waiting in queue.

6.5.3 Condor Scheduler

Condor scheduler is the daemon that receives jobs from users. The scheduler will hold jobs and information about them until it is contacted by the condor negotiator. The Condor scheduler is the daemon that will service most of the user commands (condor_submit, condor_q, condor_rm, ...).

6.5.4 Condor Negotiator

The condor negotiator is the daemon that does pairing between jobs and workers (condor startd daemons). The daemon pulls a list of available computing resources and jobs from the condor collector. It will then perform matchmaking of how to best utilize the available resources. When the negotiator finds a pairing of workers and jobs that is optimal enough it will ask the condor scheduler to send jobs to the correct worker node.

6.5.5 Condor Startd

Condor startd is a daemon running locally on each worker node. It advertises it's presence to the collector. When it is contacted by the scheduler about a job it spawns a job starter on the

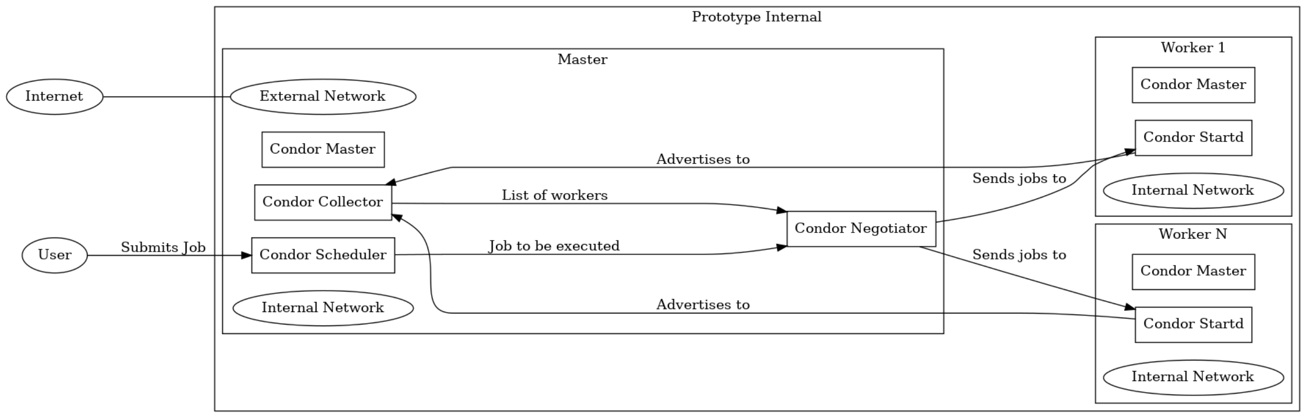

Figure 6.3: Internals of the prototype

local worker. The job starter will transfer the jobs executable and sending back the result and exit code.

6.6 Condor in the Prototype

In the prototype the master VM is responsible for running the collector, scheduler and negotiator daemons, while the workers are running the startd. This creates a clear separation of responsibilities for the different types of virtual machines. Because of this separation it is a lot easier to measure just the resources used for jobs by the system, and not include noise from the administrative tasks in the measurement.

For the various condor daemons to communicate they need a network. The prototype uses two different networks to communicate between services and with the outside world.

6.7 Networking

As a security measure the network model for the prototype uses two different networks - an internal network and an external network. The internal network is accessible by all the VMs in the pool, and allow all the services that need to communicate to do so. The internal network is considered secure, which means the internal components should not be accessible by malicious actors.

The external network only exists on the master VM, and is considered unsecure. The external network gives external access to the master, and gives the master access to the internet. Since it is unfavorable to give external actors access to everything, by default all inbound ports should be blocked, and only opened on a per-service basis. Services that benefit from access are SSH, which can allow users and operators access to submit jobs from the scheduler, or access to the collector for an external scheduler to submit jobs into the grid.

The entire stack uses only one external IP address for the master VM. The worker VMs do not require external IP addresses as they should not need to be accessed directly from outside the stack. All VMs have internal IP addresses, and all VMs know about the internal IP address of the master VM, as the condor startd requires it to connect to the condor collector running on the master.

6.8 External Prototype

The internal parts of the prototype don't do anything related to scaling the system up and down. This is done by Heat, Ceilometer and Aodh in the external system, by monitoring the internal system.

6.8.1 Scaling of Workers

The prototype should scale the amount of workers based on the average cpu load of all the workers. This is done using Ceilometer, Aodh and a Heat callback, as shown in figure 6.2. Measurement of CPU statistics is added to Ceilometer from the HOT file. Each measurement Ceilometer does is called a sample. Two alarms are added to Aodh for when the average CPU usage for the workers is over 50% and when the average CPU usage is under 10%.

When Aodh detects that cpu usage is outside of the given ratios over a certain amount of time, which in this prototype is defined to one minute (more on this in 6.8.2) it will do a callback to the appropriate URL in Heat-CFN, which will trigger scaling in the system.

When scaling up and a new worker is created it will be configured and added to the HTCondor pool, just like the initial workers when the system is first created.

When scaling down, workers will simply be removed by Heat. The Condor collector will remove the worker from the pool of online workers when it has not reported it's presence in a while. Currently there is no way in Heat to select *which* worker gets removed. This makes it possible that if one worker is executing a job, and 10 workers are idle, the machine that gets removed can be the one that is working on a job.

6.8.2 Measurement Rate

Scaling should be done as often as makes sense for your expected job length. For testing in this prototype, many small jobs that execute for seconds to a couple of minutes are run. As the jobs are relatively short measurement time was set to a single minute in this prototype, and the prototype will decide every minute if it should scale up, down or do nothing. On a larger scaled grid where jobs run several hours or even days, measurements and scaling should be done on the more appropriate scale like a couple of times per day.

6.8.3 Ceilometer Measurements

It is important that the scaling is done based on the average cpu load of the workers, and not based on the average cpu load of all VMs, which could include other VMs running on the same OpenStack platform. Ceilometer samples can be filtered to only include samples containing certain properties. Such a property can be set on a VM during VM creation which will be propagated to all samples collected by Ceilometer from that VM. In order to only consider the samples from the workers this property is set to a unique value. All queries by Aodh is set to first filter the samples by this property. This is defined more in depth in chapter 7.2.1.

6.9 Summary

To summarize, the prototype is divided into two parts - the internal and external. The internal parts submit and execute jobs. They are divided into one master and several workers. Once created, the workers will report to the master and start being used for job execution.

The external parts are configurations into OpenStack components that handle scaling and management of the prototype. Cpu load is measured on the workers and if the load is above or below thresholds workers are added or removed.

Chapter 7

Implementation

The prototype created in this thesis is made to be as close to what you would use in the real world as possible. For this reason the system was created and tested using CernVM as its virtual machine image, as this is one of the more likely virtual machine images to be found in the Cern grid setting.

7.1 OpenStack Installation

OpenStack is a highly complex application with many modules that need to communicate with each other in order to work. The installation requires configuration of all of these modules and how they talk with each other, in addition to how the modules themselves are supposed to work. In fact there are often several different modules that can fill the same role. As an example in order to get network working for the virtual machines in OpenStack one can use the old Nova-network[40], or the newer Neutron module, which until recently was named Quantum.

No complete step-by-step guides were found on how to install OpenStack from start to end. A few come close, but because OpenStack is an application with a lot of development activity, most guides require a few changes in order to work. As there were no complete guides found at the time of writing, a guide was written and attached as appendix A.

Several methods of installing OpenStack were attempted, including using the Puppet modules for OpenStack, which is documented in chapter B. With OpenStack being a fairly fast moving target with a release schedule of two releases per year[41] the version being targeted by the

Puppet module got EOLed (end of life) before a new one got released, preventing further use of the module.

7.1.1 PackStack

Due to the complexities of installing OpenStack a community project was started to make an application that is able to install and configure OpenStack, named Packstack.

Packstack uses Puppet modules to install OpenStack over SSH, but they are internal modules shipped with Packstack, and not the same modules as the ones available in the Puppetlabs repositories. It currently only supports Fedora and Red Hat Enterprise Linux derivatives, but for the scope of this thesis this is not an issue.

Packstack works well for its intended purpose, however at the time of writing there is a bug where it fails to configure the Neutron networking module properly, so this needs to be done manually[42].

Having Puppet installed before installing OpenStack using Packstack does not work either. This was attempted, but the process of installing OpenStack returns a lot of errors before failing. The easiest fix for this was simply reinstalling the system to a clean state without Puppet.

OpenStack Installation

PackStack can either receive the installation options for OpenStack through parameters on the command line, or through a configuration file which in PackStack terminology is named an *answer file*. One of the easiest ways to create such an answer file is to have PackStack generate it before it is manually edited. In this thesis the **–allinone** option is used which is a template that sets the defaults close to what is needed. The following options are being overridden on the command line:

- -os-neutron-install=y Use neutron instead of Nova-network / Quantum. On more recent options of Packstack this is the default.

- -provision-demo=n Disable the creation of a demo user

Afterwards the following options are set in the generated answer file:

CONFIG_PROVISION_OVS_BRIDGE=n This options disables the creation of an initial network. PackStack does not correctly configure a bridge network in the *Mitaka* OpenStack release [42], but once this bug is fixed it should be possible to have PackStack set up the network automatically.

CONFIG_HEAT_INSTALL=y This makes sure Heat is installed.

CONFIG_HEAT_CLOUDWATCH_INSTALL=y This makes sure the Heat Cloudwatch API is installed, which is an approximation of the Amazon CloudWatch API.

CONFIG_HEAT_CFN_INSTALL=y This installs an AWS-style API for Heat, which adds compatibility with Amazon's AWS CloudFormation template format for creating stacks. Ceilometer uses this API when an alarm is triggered.

In this configuration more compute hosts can be added at a later time - in order to do this they were added to the **CONFIG_COMPUTE_HOSTS=** section. The servers that are already installed are added to the **EXCLUDE_SERVERS=** section, which makes sure they are not reinstalled.

When PackStack is executed again it will install and configure the new servers to be added to the pool of compute hosts.

OpenStack Configuration

OpenStack is then installed by PackStack, using the answer file from above. Once the process completes, networking is changed to give the OpenStack bridge **br-ex** access to the network.

A **flat** network is going to be used by this solution, but by default OpenStack does not have this network driver enabled. A flat network is a network where all the devices are on the same subnet, which is a much simpler network topology than the default in OpenStack (vxlan), which uses multiple virtual networks and subnets to allow very complex network hierarchies. The flat network driver is enabled by adding it to the **type_drivers** parameter of the ML2 plugin configuration file for neutron at */etc/neutron/plugins/ml2/ml2_conf.ini*.

When the flat network driver is enabled a new flat network is created using neutron. This step is documented more in-depth in A.11 and includes creating a network, a subnet and a router.

A new user is created, which needs to have the role **heat_stack_owner**. This role ensures that the user can interact with Heat and be able to create and destroy stacks. Once the user is created, a new network is created by the user account. The user's network is then connected as an interface to the router belonging to the external network to give the user's network access to the internet.

DNS servers are added to the subnets. The existing network this prototype was created on only has a single DNS server and blocks traffic to external DNS servers. OpenStack requires two DNS servers, so a dummy IP (*127.0.0.1*) was used for the second server.

At this point OpenStack is functional and able to start VMs and assign real world IP addresses to them. Both the server OpenStack is installed on and external machines are able to connect to the VMs, and the VMs are able to connect to other external machines using both their IP addresses and their hostnames.

7.1.2 Changing the Ceilometer default polling interval

By default Ceilometer polls for CPU usage every 10 minutes. That is a bit slow for this stack, as it would take at least 10 minutes, and up to 20 minutes of time before any change to the load on the system would be reflected by the scaling so this will be changed to 1 minute.

In a real world scenario 10 minutes may even be too high, and needs to be set to a reasonable number for the system it will be used on.

The change is done in the **/etc/ceilometer/pipeline.yaml** file. On the section with the name **cpu_source** the interval is changed from 600 to 60.

7.2 Creating a Heat stack

Heat acts as the brain of the cloud where it decides what to do with the stack based on templates it receives when a new stack is created. If Heat is the brain, then Telemetry is the eyes of the system, doing all the monitoring, collection and processing of the data. Telemetry enables setting of alarms when certain conditions are met, for example *if average CPU usage across all machines in the stack is over 50% for at least 2 out of the past 3 minutes.* When an alarm is triggered Ceilometer does a callback to the heat-cfn-api which acts on the alarm.

The Heat Orchestration Template (HOT) describes which resources should be provisioned, how they should be initialized and which dependencies there are between them.

The HOT files created for this project uses the **parameters**, **resources** and **outputs** sections available in the HOT format.

Parameters The parameters in the main HOT file are the ids for all the public and private networks and subnets in use by the stack, and the name for the ssh key that should be able to log into the servers. This makes the stack portable and usable on different systems that don't have the same ids and names for the networks and keys available.

Resources The resource section is where most of the logic is. It includes many resources that will be explained in more detail in 7.2.1. The resource section describes all the dynamic parts of the VMs, but the implementation of the VMs is split into separate files for clarity, and all dynamic parameters are passed to them through the parameters section.

Outputs The outputs section describes values that are exported from the stack. The outputs are the URLs for scaling the stack up and down, which enables an operator to override the automatic scaling and manually scale the stack to the size wished. The other outputs is the stack-key which all the metadata from the workers is tagged with in order to enable filtering of this data and the query Ceilometer uses to gather statistics about the VM metrics, which is available for debugging purposes.

7.2.1 HOT resources

condor_master

The condor_master resource is a static VM that always exists in the stack. The main logic of the resource is in the separate file master.yaml explained in more detail in 7.3.1. When the condor_master resource is created it runs a user-script which installs and sets up the **ht-condor** environment before using a callback to the wait_handle resource from 7.2.1 indicating that the stack can now start making worker VMs.

This VM has a private ip address that is accessible from everything within the stack, but unlike the other VMs the condor_master also has a public ip address that is accessible from everything outside the stack.

wait_handle and wait_condition

Wait conditions in Heat are resources that wait for an event to occur before it gets the status of *completed*. This means that any other resources that have a dependency on the wait condition will also wait for the event to occur before starting to be built.

The most commonly used event for wait conditions is a wait handle which can be interfaced to from outside of Heat by posting to its URL. In this prototype the wait handle waits for a single event, which is the **condor_master** POSTing a request to the handle URL before releasing the wait condition to be completed. The auto scaling group that contains the worker VMs has a dependency on the **wait_condition**, which ensures no worker VMs will be created before the master VM is created and ready.

static_port and static_floating_ip

For the **condor_master** to be available from outside the stack it needs to have a floating ip address. This is done by creating a **OS::Neutron::Port** resource and attaching to the server. This type of resource acts as a network port attached to the server with a dedicated internal ip address that can access other servers within the stack. A **OS::Neutron::FloatingIP** is then attached to the network port with a publicly accessible IP address, making the **condor_master** accessible from outside the stack and allowing global internet access to it.

Scale up and down policies

The HOT implements two scaling policies - the **scaleup_policy** and the **scaledown_policy**. The policies are interfaces to the stack that allows adjustments of values within the stack, in this case the stack size, to be made. The scaling policies interfaces are accessible through URLs that are created upon stack creation. For manual control of the stack these URLs are also exported as **outputs** and can be POSTed to by an operator to manually scale up or down the stack.

The scaling policies, when triggered, are set up to increase or decrease the amount of VMs by one with a cooldown of 60 seconds before the policy can be triggered again.

CPU alarms

The scaling policies themselves just implement scaling, they don't actually execute any scaling based on various factors. For the stack to start scaling automatically, two resources of type **OS::Ceilometer::Alarm** are used, named **cpu_alarm_high** and **cpu_alarm_low**. These resources represent actual alarms in Ceilometer (Aodh) that Heat creates as part of the stack creation. These alarms are created to trigger an action when the alarm is triggered, in this case sending a POST request to the URLs of the scale up and scale down policies. The alarms are evaluated once every minute, which will trigger the scaling policies once every minute if needed.

The alarms use metadata matching to only use the samples tagged with the correct metadata, which in this case is the global stack id.

The alarm **cpu_alarm_high** is set up to trigger if cpu utilization was above 50% in the past 60 seconds and will call the **scaleup_policy**.

The alarm **cpu_alarm_low** is set up to trigger if cpu utilization was below 10% in the past 60 seconds and will call the **scaledown_policy**.

Auto scaling group

An auto scaling group is a resource that contains zero or more of another resource and can change the number of instances of that resource at runtime. In this case the resources are instances of the worker nodes running **CernVM 4**. The auto scaling group has a dependency on the **wait_condition** resource and as such will not be created before the **condor_master** is running and has finished initialization.

The properties for the auto scaling group includes a cooldown variable which decides how often the stack can scale up or down, minimum size, maximum size and **desired_capacity** which represents the current amount of VMs.

The desired_capacity variable is interesting as it represents the current stack capacity. This means that upon stack creation this will be the number of worker VMs created, but it also means that if an operator wishes to set the VM count to a certain number at runtime the operator can update this variable in the template and *update* the stack, using the **heat stack-update** command, with this new template which will set the internal VM count variable to what it is in the new stack template and create or destroy VMs in order to reach that number.

The auto scaling groups resource is the HOT template for the worker VMs, named **cernvm4.yaml**. The CernVM template requires a parameter for which SSH key to inject into the image in order to allow a user to log in to a running instance and also which metadata the VM should have.

There is a special metadata value named **"metering.stack"** that can be set on a VM that is added as a property, or a tag in all samples gathered by Ceilometer about that VM. Tagging samples in this way allows Ceilometer queries about just the VMs in the auto scaling group instead of querying about all the VMs in the stack, or even all VMs running on the OpenStack system.

When a new stack is created using heat a unique **stack_id** is created, which is the global id of the stack and is accessible via **OS::stack_id** in the HOT file. When Heat creates VMs in an auto scaling group it creates internal sub-stacks for each VM, which means that accessing the **OS::stack_id** variable from within the stack template of that VM will return the **local stack id** at runtime, which is the id of the substack, and not the **global stack id**. The **global stack id** is not available from the sub-stacks at runtime, and to be able to tag the metadata with the correct stack id the solution chosen in this project was to simply send the correct id in to the substack using a **parameter** in the HOT file for CernVM4.

The VMs are in this way tagged with the value of **OS::stack_id** which is the global id of the stack so Ceilometer can query the auto scaling group for information.

7.3 VM Templates

7.3.1 master.yaml - condor_master

The master.yaml HOT file takes four parameters:

metadata The metadata to add to the VM. This is set to **'None'** as it isn't used for anything; but is available if it is going to be required later.

network_port The network port to be used. This is how the **static_port** from 7.2.1 is added to the VM.

user_data The user data is a string that will be sent into the VM and then in this case be executed by CloudInit which is available on most cloud images, including the CernVM 4 image. For the condor master the user data passed to the template installs wget in order to download and install the **htcondor-stable** repository and signing key before installing the **condor-all** package through the standard yum package manager. Condor is then configured to allow remote workers, and then started before the VM finally reports success through the **wait_handle** defined in 7.2.1.

keyname This is the name of the SSH key in Nova to allow logins to the VM.

The HOT file itself is fairly simple, containing just two main resources which is the **server** which is an instance of **OS::Nova::Server** and represents the actual running VM instance, and a **volume** resource of type **OS::Cinder::Volume** that is the disk image that is copied and added to the **server** before it is started.

The **server** resource consists of all the properties the VM should have, including the **flavor**, which is a set of virtual hardware the machine should have, the networks, the userdata, metadata and the key name for SSH access.

The image name could be set as a parameter to allow starting from arbitrary images.

Before use the image needs to be downloaded by **Cinder** from **Glance** and be converted to a **raw** image. This process can be time consuming, especially for larger images, which can lead to slow stack creation. This issue is discussed more in depth in 7.5.2.

7.3.2 cernvm4.yaml - condor workers

At first CernVM 3 was used, but there was an issue with Condor on this version of CernVM that prevented it from running any jobs. This issue is described more in detail in 7.5.8. For this reason CernVM 4, which is in beta, is used instead.

The cernvm4.yaml HOT file takes four parameters:

metadata The metadata to add to the VM. This is a json string sent to Nova to be applied to the virtual machines. The metadata is set to the json string **'{"metering.stack": {get_param: "OS::stack_id"}}'** where metering.stack is a special property. The value that **metering.stack**

is set to will be appended to all samples gathered by Ceilometer and available as **metadata.user_metadata.stack** in those samples. That means that you can *tag* all samples from a set of VMs and have Ceilometer create statistics based on only those samples. Making the alarms defined in 7.2.1 trigger only on data from the worker VMs is done using this **metering.stack** attribute.

user_data The user data is a script that will be sent into the VM and then be executed by CloudInit which is available on most cloud images, including the CernVM4 image. The user data script configures Condor to use the **Condor master** as the master before starting the service.

keyname This is the name of the SSH key in Nova to allow logins to the VM.

The cernvm4.yaml HOT file contains many of the same things that the master.yaml HOT file does with a few differences.

The cernvm4.yaml's disk volume is smaller at just 1GiB as opposed to the 2 GiB of the master, which makes starting and destroying VMs slightly faster. This makes sure the speed of scaling up and down using CernVM is more rapid than if a larger image was used.

The flavor used in the image is smaller than in the master, as the workers requires less memory in order to operate because they run fewer applications in the background.

7.4 Condor configuration

On CernVM Condor has a default configuration that needs to be modified in order to be able to connect the worker nodes to the master node. The Condor configuration system includes the file **/etc/condor/condor_config.local** at the end of the default configuration file, which allows settings to be overridden by placing an override in this file. The overrides in use are:

Servers	Setting name	Value	Purpose
Both	ALLOW_WRITE	*	Disables most ip and domain checks for writes.
Both	ALLOW_READ	*	Disables most ip and domain checks for reads.
Worker	CONDOR_HOST	STATIC_IP	Points to the condor master server. STATIC_IP is replaced by the internal ip address for the condor master.
Both	TRUST_UID_DOMAIN	True	Disable UID domain check. Okay to disable as the prototype is not accessible from the internet.
Both	ALLOW_NEGOTIATOR	*	Allow negotiator from any ip
Both	ALLOW_NEGOTIATOR_SHEDD	$(ALLOW_NEGOTIATOR)	Sets checks to the same as ALLOW_NEGOTIATOR (from any IP).
Both	UPDATE_COLLECTOR_WITH_TCP	TRUE	Uses TCP instead of UDP when sending updates to the condor_collector.
Worker	START	TRUE	Condor job starter will run jobs when this is set to TRUE

Both	HOSTALLOW_WRITE	*	Allow all IPs to do everything.
Worker	DAEMON_LIST	MASTER, STARTD	Which daemons should be run on the worker node. Only job starter and master.
Master	DAEMON_LIST	COLLECTOR, MASTER, NEGOTIATOR, SCHEDD	Which daemons should be run on the master node. Everything except job starter.

There are a few security checks, like the **ALLOW_*** settings, that are disabled for simplicity sake. These do things like validate that requests come from the correct domains and IP addresses. Since the entire prototype runs on a dedicated subnet without access to the internet this isn't a security risk as the only possible place any requests can come from is within the subnet.

7.5 Solution Issues

There are some issues that appeared during creation of this project. Some of these, like the Heat scaling issue, are due to bugs in code, that cause issues constantly. Other issues, like the reboot issues and database issues may be caused by misconfiguration by Packstack. *Anshu Awasthi* writes in *Comparison of OpenStack Installers*[38] that Packstack deploys OpenStack with minimal configuration and *"The OpenStack deployed from Packstack is not stable enough to handle the data center requirements."*.

7.5.1 Heat Engine Scaling Issue

There is an issue with the version of heat-engine in use by this prototype where some times the heat engine will get permanently unable to scale a stack up or down[43]. The issue is caused by python interpreting the string *'scaling_in_progress'* as a date and causing the entire query to throw an exception. When this happens there is no consistent way to solve it. Some times recreating the stack would solve the issue for a while, and other times it did not. The bug is fixed in versions 7.0.0.0b1, 6.1.0 and 5.0.2 of the heat-engine, but in the prototype for this project version 6.0.0 of the heat-engine is used; which contains the bug. The issue is worked around by applying the patch that solved the issue on the later versions to the installed version and force-recompiling the related python packages. The patch is found here: `https://git.openstack.org/cgit/openstack/heat/diff/heat/scaling/cooldown.py?id=080ace0054682494134a5bca5921fe9`

7.5.2 Slow VM creation by Heat

When Heat is going to create a new VM it first asks Cinder to download the entire image from Glance's backing store. Cinder then runs qemu-convert on it in order to validate the image and convert it to the raw format that can be written straight to a volume. Then it copies the converted image into lvm to create a valid cinder volume from it. This entire process can take several minutes for a large image and is done each time Heat creates a new VM, making the process of creating new VMs potentially slow.

If Heat is not in use then the solution is to create a template volume with Cinder which is already converted (and usable), and instead of creating a new one for each VM just copying the template volume as it is a lot faster. At the time of writing I did not find a way to do this with Heat, but the developers are aware of the issue and all the overhead that is currently involved so it is an issue that will hopefully be fixed in the future.

7.5.3 Error 500/503 causing issues and load averages

At random times calls to the OpenStack APIs return **error 500** or **error 503**, causing the call to fail. The conditions in which this occurs were not deterministic, but anecdotally it *feels* like it occurs more often when there is a high load average on the system.

The Unix load average stat is *the amount of resources that one or more processes are waiting for*. If the load average is higher than the amount of processors in the system the system can begin to run significantly slower. During stack creation by Heat the load average will increase significantly - at times as high as 15.

When the load average increases significantly above the amount of processor cores it seems a lot more calls to OpenStack fail than otherwise.

This is an issue as Heat uses a lot of calls to the other OpenStack APIs when creating a new stack, and if a single one of these calls fails the stack creation fails and will need to be restarted. The same can happen when deleting a stack, in which case the operator needs to log into Heat's database and make manual changes before attempting the stack deletion again.

7.5.4 Potential memory leaks

The memory usage by OpenStack slowly creeps upwards and while OpenStack uses around 6GiB of memory when started it will increase and after a few days of usage it will be at more than 8GiB. After having OpenStack running for over 2 months the OpenStack services used 11GiB of the 12GiB of memory available on the master with no VMs running. It seems the amount of *error 500/503*s increase with the memory usage as well, but this could be because of increased swap usage when a lot of memory is in use.

7.5.5 Reboot issues

When the entire system is rebooted **MariaDB** fails to start, potentially because it attempts to start too early in the boot process before networking is operational. This causes several OpenStack services that depend on the database to fail to start as well, including **neutron**, which handles networking.

In one instance **MariaDB** did start and run, but did not allow connections, nor was the process recognized by Systemd as running. Attempting to do a restart on **MariaDB** during this event started an additional MariaDB process that could not lock files and crashed shortly thereafter. After several unsuccessful attempts of stopping the process using the **Systemd systemctl** interface, the process was killed manually followed by successfully starting MariaDB. This allowed the other OpenStack processes to finally start.

7.5.6 Database issue

During a reboot of the server containing the database application MariaDB the logs showed that MariaDB did not shut down in time and was killed by the system. When the system was booting again MariaDB showed that there was one ongoing transaction at the time of shut down with almost 300,000 updates. MariaDB proceeded to automatically roll back the transaction before starting, which took around 11 hours before the database server started and started accepting connections.

Due to the huge amount of updates in the transaction it is suspected that some component of OpenStack opens a new transaction and proceeds to do updates in this without ever committing the updates. This could explain both the memory leaks from 7.5.4 and the reboot issues from 7.5.5.

7.5.7 Neutron router disappearing

After a reboot OpenStacks network service Neutron did not start the router namespace required for routing traffic. This namespace is required in order to route traffic to/from the VMs. All settings for the router were verified to be correct and the service, as well as the entire machine attempted restarted several times without the machine recovering from the issue. The solution for this issue is to delete the entire router and add it again, including setting the gateway and adding interfaces to it, which will cause it to start correctly again.

7.5.8 CernVM 3 issues

The project used CernVM 3 at the start, but Condor has an issue where the job starter on CernVM 3 would make a function call to **strlen_SSE4** in **glibc** which caused a segfault. Often this type of issue occurs when the software in question, which in this case is Condor, is compiled on a system with a more recent version of glibc with some newer features and then attempted to run on a system with an older version of the library. It may then attempt to make a function call to a function that exists in the newer version of the library, causing a segfault as it doesn't exist in the older version.

If this is the root cause of the issue is unknown. The fix was to use CernVM 4 instead, which is in beta. CernVM 4 was a drop-in replacement for CernVM 3, and the only change required was to change the image used to the new CernVM 4 image.

Chapter 8

Evaluation

To evaluate the prototype implementation, stacks were deployed to the test environment several times. In total over 220 stacks were deployed, and the longest running stack ran for over three months, executing jobs and scaling sporadically. Various tests were used to verify that the various functional requirements were fulfilled.

8.1 Tests

This section describes the tests that are used to verify that the functional requirements are fulfilled. The tests were run multiple times to uncover potential issues that may not show up on every run.

Job submission and execution

An executable was created, which takes a single argument and prints out the result to stdout. A job was then created that executes this executable 1,000 times with the job number as the first argument to the executable. The job was submitted and the resulting outputs verified. This ensures that both submitting a job and executing it works.

Recreatable environment

The stack should be created and destroyed several times, and jobs executed on the stacks to verify that job execution works. The stack will be recreated between each test, and tests will be run multiple times to ensure consistency.

Measurements will also be done to establish an estimate of how long the stack takes to create, and how often the creation fails. As grid jobs often run for several hours, a creation time of under an hour would be acceptable.

Automatic Scaling

An executable, which takes a single integer N as an argument and performs a computationally intensive task for N seconds was created. A job was then created to run multiple instances of that executable with long execution times.

For this test N was set to **600**, so each job simulates work for 10 minutes. 72 jobs were queued for a total work-time of 720 minutes, or 12 hours. The expected result is that new worker nodes will be added and the test will take less than 12 hours.

The **scaling factor**, which represents the **average amount of nodes executing jobs** can be found by taking the total work time (720 minutes) and dividing it by the total time used by the test. Since the maximum amount of workers for the system is set to 7, the theoretical maximum scaling factor is 7. If no scaling occurs the scaling factor is 1.

When all the jobs have been executed it shall be verified that the system removes worker nodes that are no longer required.

Manual control

A stack will be created. The template that was used to create the this stack is then modified and the stack updated to verify that the operator can change the amount of workers and scaling properties of the stack at will.

For this test the stack will initially start with one worker, a minimum of one worker, and a maximum of seven workers. The stack will then be updated to have seven workers with a minimum of three workers. It shall be verified that the stack spawns six more workers for a total of seven, before it scales down to the new minimum of three workers.

Job completion during down scaling

This test will have a long-running job with an execution time of 10 hours, which will running for the entire duration of the test. The system will then be scaled up and down repeatedly while the long-running job is executing. The **automatic scaling test** determines that the system uses around 20 minutes to fully scale up or down.

To make the system scale up and down N-1 additional jobs, where N is the maximum amount of workers will be added each hour into the test. The jobs will execute for the same duration that the system uses to scale up (20 minutes), to make sure the system scales up to it's maximum. Since there are N workers, and a total of N jobs, all jobs will be started at 20 minutes into the test. The jobs run for 20 minutes and should complete at most 40 minutes into the test, at which point the system will scale down again, which takes another 20 minutes. At 60 minutes the system should be scaled down as far as it's going to be with the load on the system, at which the shorter jobs will be added again to make the system scale back up. The N-1 shorter jobs will be added hourly until the tests complete, ensuring all the worker VMs will be destroyed at least once during the test.

For this test changes have been made to the prototype in order to make it scale down more aggressively. In it's default settings the system would only scale down when average CPU usage of the workers is less than 10%. With a maximum of 7 workers, a single worker with 100% cpu load could make sure the average was above the 10% threshold and nothing would be scaled down. The threshold for this test has been set to 30% to make sure the system scales down when intended.

8.2 Results

This section presents the results from the tests.

Job submission and execution

This test completed without any issues. All jobs executed and the results (stdout and return value) were transferred back to the submission machine correctly. Three workers were used, and all 1000 jobs were executed in 1 hour and 48 minutes. This means that each worker did about 3 jobs per minute (3.08), giving an average time of 20 seconds to retrieve the job, execute it and send back the result. Since the job only prints out it's first argument and executes in milliseconds, the overhead of this job (time to retrieve the job, upload the result, and wait for the next job) is 20 seconds.

Recreatable environment

During testing there were times where the stack failed to be created or destroyed. This can happen when Heat asks one of the other OpenStack services to perform an action, like creating a new VM, and that service's API returns a HTTP 500 or 503 error. This is documented more thoroughly in 7.5.3.

It is mentioned in 6.2.3 that Packstack does have some stability issues, and this error may be an effect of one of them. Further testing is required to find out if this is the case.

The time used to create the stack, and how often stack creation failed was measured. A summary of the measurements can be found in table 8.1. A total of 91 measurements were done. The time creation took was measured on the 72 stacks that were created successfully. The failed creation attempts accounted for 19 of the samples.

Figure 8.1 displays the time used to create the stack during the run. The measurements were done with 30 second resolution, which means after a measurement finished there was a 30 second pause before the next one started.

Number of creation attempts	91
Minimum creation time	208s
Maximum creation time	504s
Average creation time	234.2s
Number of successful creations	72
Number of failed creation attempts	19
Percentage of failed creation attempts	20.9%

Table 8.1: Stack creation statistics

Table 8.1 shows that the runs that completed had an average creation time of 3 minutes and 54 seconds, which is well within an acceptable range.

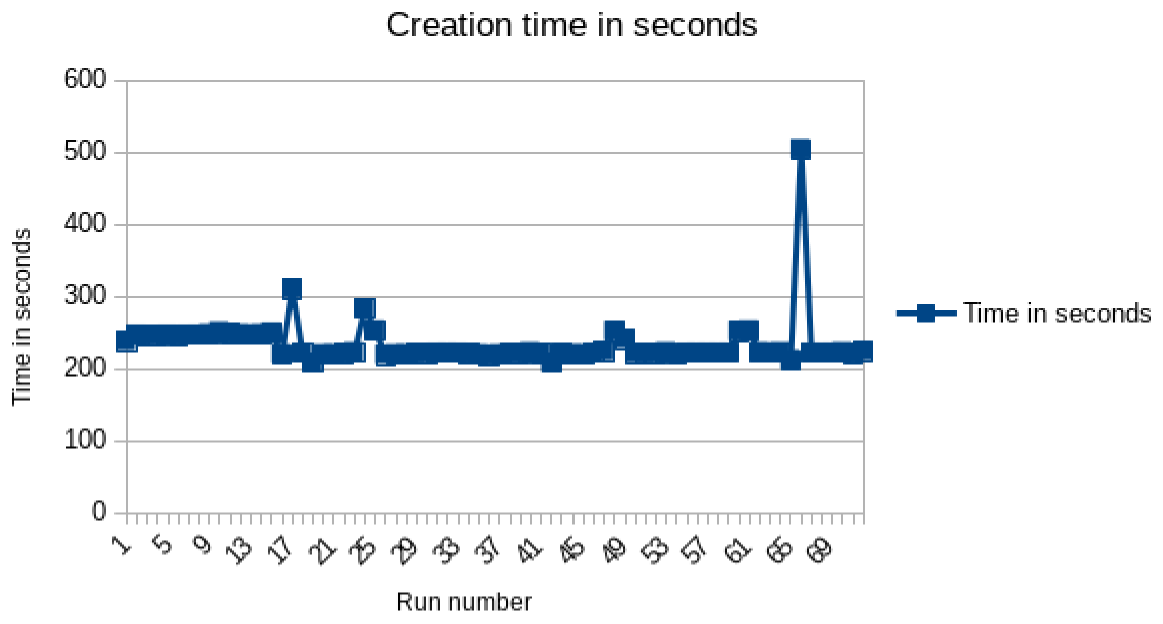

Figure 8.1: The time in seconds to create the stack.

The stack creation failures and the slowly increasing memory documented in 7.5.4 may be related to configurations that Packstack performs during installation, in which case this is not an issue in solutions that install OpenStack by other means; but testing is required to see if this is the case.

Automatic Scaling

The job was queued, and finished 120 minutes later. This gives a scaling factor of 720 minutes / 120 minutes = **6.0**. The theoretical maximum for the test was 7, but some time is used before the system is scaled to it's maximum, and there is also some overhead between jobs.

When the test concluded it took an additional 19 minutes before the system had scaled down to the minimum of 1 worker node.

An additional longer test was run. The total work was 720 jobs, of 10 minutes each for a total work time of 7200 minutes (120 hours). The same parameters were used for the job, with a maximum scaling factor of 7. The run finished in 1093 minutes (18 hours, 13 minutes), giving the run a scaling factor of 7200/1093 = **6.587**.

There are two main factors that impact this score to cause it to be lower than perfect. There is some time between each job where the batch system will upload the result of the previous job before pulling the next job. Another factor is that the scaling of the system is gradual, and it takes some amount of time to start new VMs.

The scores being more than or equal to 6.0 means that the system scaled up rapidly when it detected an increase in the load on the system.

Manual control

The stack was initially started with the values of minimum workers set to 1, maximum workers set to 7 and current workers set to 1. The stack created successfully and stayed at one worker as expected. The template was then updated to have current workers set to 7, and minimum workers set to 3. When the stack was updated with the new template it immediately spawned six new workers for a total of seven before eventually scaling down to three workers where it stayed.

This verifies that an operator can have control over the amount of workers and properties of the scaling at will.

Job completion during down scaling

The test was started, and jobs were added automatically, causing the system to scale up. Once the shorter jobs had completed the system started scaling down again. The very first VM that got removed was the one running the long job. 80 minutes after the test started, around 40 minutes after the node running the long-running job got removed, the job got re-queued on another node, before that node was removed as well. The author terminated the test after 7 hours, as the worker running the long job kept getting removed and started over.

The result is that this test failed, and Heat removed the VM executing the long-running job. An issue with Heat, is that there is no way to define *which* VMs should be removed when scaling down; and VMs still running jobs can be scaled down. Heat will always remove the oldest VM.

After the test, the environment was left to scale down on it's own. When the author checked on the system the next day, it had scaled away all the workers, even though the template is configured to leave a **minimum** of one worker. The test was run again, but the issue could not be replicated.

An environment can be created with a very long running job, in the ballpark of one day, and have batch jobs to be executed every hour, causing the system to scale up during batch processing and down when the processing completes. In this environment the long-running job would never finish as the worker processing it would always be scaled away before the job completes.

8.3 Validity of Results

The main motivation behind the development of the prototype is to determine if such a solution would be viable to implement on a real-world grid site. The evaluation criteria were developed, and the solution tested to uncover potential issues with such an implementation. It should however be noted that tests were run on a small-scale test system, and the results should be treated as such. There may be issues that would occur on a larger scale that will not be uncovered by running small-scale testing.

On a real-world grid platform there would likely be a middleware, like AliEn, in use; and such a middleware integration was not tested in this thesis. AliEn has a built-in option for using the HTCondor batch system and submitting it's job agent there, but this has yet to be tested on this system.

The jobs executed in the tests were locally developed, and no real-world grid jobs were used in testing. The jobs simulate integer-based computational work, but this may not have the same computational characteristics that real-world grid jobs have.

It should also be noted that the test for automatic scaling makes the assumption that a job will cause an increased computational load on the system. If a job did not use a lot of cpu resources, for example because it was sitting idle and waiting for data to be ready, the test would likely yield a different result.

The OpenStack environment could also be a factor in the results. In a real-world scenario there would likely be an enterprise-grade OpenStack installation, and Packstack would not be used. The author suspects some of the issues encountered during the development of the prototype is because of the configurations made by Packstack, and would not be present in a more optimal configuration. Another difference is that the test OpenStack system only had a single user on at any time, while a larger system will likely have many more.

The results from the tests are still valid, for the things tested; but as described above there are limitations to *what* has been tested.

8.4 Discussion of Results

The **job submission and execution**, **recreatable environment** and **automatic scaling tests** worked as intended and provided results that for the use cases that were tested the prototype works well. The **manual control** test shows that an operator can have full control over the system, which enables use of the prototype, even in scenarios where the automatic scaling must be disabled. This also ensures that a solution like this prototype can be used for temporarily taking advantage of resources, even without scaling - as long as the operator destroys the stack once it is no longer required.

The **job completion during down scaling** test failed as expected, as there is no way to select *which* VMs Heat should remove when scaling down and the VM running the long job got removed. Heat scales in a FIFO (first in, first out) order. This does limit the scenarios this particular solution should be used in, and it is possible that for some real-world applications the automatic down scaling should be disabled and be performed manually instead.

Chapter 9

Conclusion

Cloud computing provides technologies like scaling and virtualization, that can be used by computational grids to improve security, enable elasticity and make the workers in the grid homogeneous. Elasticity can ease the process of adding and removing nodes, which can give the grid access to additional resources that may just be temporarily available.

The goal of this thesis was to create a prototype implementation of a system that leverages the technologies provided by cloud in order to create an elastic homogeneous grid environment. A prototype was created by using the OpenStack services Heat and Ceilometer to provide elasticity, and HTCondor to provide computational job execution. A set of functional requirements were defined and the prototype tested to see how it fulfills the requirements. The environment creation, job execution, automatic scaling and manual control tests succeeded, but one test that was supposed to check that no nodes currently running jobs would be removed by the elasticity failed; as there is currently no way to choose which node gets removed when scaling down.

Reviewing the functionality achieved and the results from testing, it can be concluded that the technology looks promising, but further work and testing is needed to determine if some of the issues encountered can be solved.

9.1 Further Work

This prototype used the OpenStack service Heat to provide elasticity to the platform developed. There is another, newer OpenStack service that can provide automatic elasticity named Senlin, which is documented more in chapter 3.4. Senlin seems to provide more control over how scaling is done than Heat, and a solution based on Senlin instead of Heat could fulfill the requirement that failed in this prototype.

Integrating this solution with a grid middleware like AliEn and running real AliEn jobs is something else that should be tested and verified to work.

There are also several seemingly random unexplained stability issues that have been encountered during development and testing. Some are suspected to have occurred because of configurations made by Packstack, but additional testing should be performed to determine the overall stability of the system.

Appendix A

Installation of OpenStack

A.1 Background

OpenStack is installed on a clean CentOS 7 install without any work done on it. After the install the server will run OpenStack with a single user account, and be able to start VMs that can be accessed from other machines in the network, and is able to access the internet.

Unless otherwise specified all commands will be ran as the **root** user.

A.2 Requirements

After a completed install with OpenStack running the server will be using around 8GiB of memory, and additional memory for each virtual machine that is running. The install was tested on a machine with 6GiB of memory (and 6 GiB of swap), and it was functional, but very slow at times; so at least 10 GiB of memory is recommended.

A continuous block of ip addresses available on the network that will not be allocated by the DHCP server. At least 3 is required - two plus one for every VM you want to have its own address.

A.3 Setting up the basic environment

Installing tmux or screen is recommended to be able to do things in parallel and be able to recover the terminal session if you happen to be disconnected from the machine; it is not required however.

I use the text editor *vim* in this setup, but feel free to replace it with your favorite editor.

Install vim and tmux using the following command - leave out tmux if you wish to not use it and start a new tmux shell.

```
$ yum install -y vim tmux
$ tmux
```

Listing A.1: Installing vim and tmux

The next step is updating all the packages on the CentOS install.

```
$ yum update -y
```

Listing A.2: Updating CentOS

Press **ctrl+b** and then **d** to detach from the tmux instance and let it run in the background.

A.4 Setting a static IP address

OpenStacks default network stack does not work well with **NetworkManager** which is the default network manager in CentOS 7, so we will set a static IP address. Network information is in **/etc/sysconfig/ifcfg-<interface name>**. On the system this thesis is using this is **/etc/sysconfig/ifcfg-enp1s0**.

Edit the file with your text editor of choice to contain static IP information. Update the IP address to what is required in your network.

```
BOOTPROTO=static
IPADDR=10.0.0.218
DNS1=8.8.8.8
NETMASK=255.255.255.0
GATEWAY=10.0.0.1
NAME="enp1s0"
DEVICE="enp1s0"
ONBOOT="yes"
```

Listing A.3: Content of /etc/sysconfig/ifcfg-enp1s0 when setting static network

The tmux session from earlier is attached back to in order to see when the updates are complete.

```
$ tmux attach
```

Listing A.4: Attaching to tmux

When the updates are completed the network service is enabled and the NetworkManager service is disabled.

```
$ systemctl disable NetworkManager
$ systemctl enable network
$ systemctl stop NetworkManager
$ systemctl start network
```

Listing A.5: Disabling NetworkManager and enabling network

The machine is then rebooted in order for kernel updates to take effect.

```
$ reboot
```

Listing A.6: Rebooting

If the network still works the system is ready to continue the installation.

A.5 Installing Packstack

To install Packstack the system first requires the rdo software repository. It should be noted that installing third party software repositories has potential security implications as it trusts that the maintainers have good intentions. An ill-meaning maintainer could create a package that places a backdoor in your system.

Installing the rdo repository is done by the following command.

```
$ yum install -y https://rdoproject.org/repos/rdo-release.rpm
```

Listing A.7: Installing the rdo repository

Then we enable the OpenStack repository that contains the Mitaka version of OpenStack.

```
$ yum install -y centos-release-openstack-mitaka
```

Listing A.8: Installing the OpenStack-Mitaka repository

There are a few packages that have new versions in the repos that were enabled, so the system is updated again before the **openstack-packstack** package is installed.

```
$ yum update -y
$ yum install -y openstack-packstack
```

Listing A.9: Updating the system and installing Packstack

A.6 Creating the Packstack answers file

Packstack has a lot of options available, and rather than providing all of them through the command line we will create an answers file containing all the options needed as this allows recreatability of the setup later.

First go to the /root directory and then generate the initial answers file.

```
$ cd /root
$ packstack --gen-answer-file answers packstack --allinone --os-neutron-install=y --
    provision-demo=n
```

Listing A.10: Generating the answers file

There are a few options here - their meaning is the following.

- -gen-answer-file answers.conf Instead of running the install, generate an answers file for it into answers.conf

- -allinone A large set of default options for installing on just one machine (the one we are on).

- -os-neutron-install=y Install neutron instead of the other networking modules. This may not be required depending on the packstack version.

- -provision-demo=n Disables installation of the demo user and demo project

There are a few more changes needed to the answer file. Find the following lines in the answers.conf file using your text editor.

```
CONFIG_HEAT_INSTALL=n
CONFIG_PROVISION_OVS_BRIDGE=y
```

```
CONFIG_HEAT_CLOUDWATCH_INSTALL=n
CONFIG_HEAT_CFN_INSTALL=n
```

Change the lines into the following.

```
CONFIG_HEAT_INSTALL=y
CONFIG_PROVISION_OVS_BRIDGE=n
CONFIG_HEAT_CLOUDWATCH_INSTALL=y
CONFIG_HEAT_CFN_INSTALL=y
```

The explanation for the options is the following:

CONFIG_HEAT_INSTALL Sets wether Heat is installed or not. This is an essensial component of this thesis, and therefore enabled.

CONFIG_PROVISION_OVS_BRIDGE If this option is enabled a default network will be created. Here the network will be created manualy anyway, so the default is removed to avoid confusion.

CONFIG_HEAT_CLOUDWATCH_INSTALL If this option is enabled a cloudwatch-like API for Heat is installed.

CONFIG_HEAT_CFN_INSTALL This installs an AWS-style API that is used by Ceilometer when an alarm is triggered.

This disables installation of the default network and enables installation of the **Heat** OpenStack module as well as the Cloudwatch and CFN APIs for this. Those are all the changes required to make to the answers.conf file, and installation of OpenStack is now ready.

A.7 Installing OpenStack

```
$ packstack --answer-file answers.conf
```

Listing A.11: Installing OpenStack

That's all. Packstack installs OpenStack and the **- -answer-file answers.conf** parameter tells packstack where to find all the answers it requires for how to do so. This process will take a long time depending on the hardware in use - often as much as 30 minutes.

When the process completes there should be no errors.

A.7.1 Enabling the API services

When OpenStack was installed the heat-cfn-api service was not installed on the system this was created on so make sure it is installed by running the following.

```
$ systemctl enable openstack-heat-api.service openstack-heat-api-cfn.service openstack-heat-engine.service
$ systemctl start openstack-heat-api.service openstack-heat-api-cfn.service openstack-heat-engine.service
```

Listing A.12: Enabling and starting all the Heat services

A.8 Setting up the network interfaces

There is now a few new network interfaces, including the bridge-exit interface (br-ex). For OpenStack VMs to be able to communicate with the outside world we will need to give this interface networking information.

Open up the **/etc/sysconfig/network-scripts/ifcfg-br-ex** file in your text editor and insert the following text before saving it.

```
DEVICE=br-ex
DEVICETYPE=ovs
TYPE=OVSBridge
BOOTPROTO=static
IPADDR=10.0.0.218
DNS1=8.8.8.8
NETMASK=255.255.255.0
GATEWAY=10.0.0.1
```

Listing A.13: /etc/sysconfig/network-scripts/ifcfg-br-ex

Change the values to match your own network, just like was done in A.3.

Edit the file describing your main interface - on my system this is **/etc/sysconfig/network-scripts/ifcfg-enp1s0** - and replace all the text with the following.

```
TYPE="OVSPort"
```

```
NAME="enp1s0"
DEVICE="enp1s0"
ONBOOT="yes"
DEVICETYPE=ovs
OVS_BRIDGE=br-ex
```

Listing A.14: /etc/sysconfig/network-scripts/ifcfg-enp1s0

Now reboot the machine and check that it still has internet. If the machine isn't able to connect to the internet check the two files edited in A.13 and A.14.

A.9 Prerequisites for the network

OpenStack has several network types. The default type is **vxlan** which creates a new vlan for each user, but would complicate the network setup, so instead the **flat** network type will be used. A flat network is a network where all the VMs reside on the same network and can communicate with each other and the outside world.

First the **flat** network driver needs to be enabled. To do this the **/etc/neutron/plugins/ml2/ml2_conf.ini** file is opened with a text editor. The type_drivers line looks like the following:

```
type_drivers = vxlan
```

Add «**,flat**» to the end, so it looks like the following:

```
type_drivers = vxlan,flat
```

For the changes to take effect Neutron needs to be restarted.

```
$ systemctl restart neutron-server
```

Listing A.15: Restarting Neutron

A.10 Authenticating with OpenStack

All the OpenStack command line clients need to authenticate before performing any operations. Authentication can either be done through the command line by adding several parameters

(Username, password, authentication url, authentication type) to each command, or through environment variables.

Packstack has already created a file containing all the required environment variables needed by the OpenStack clients in **/root/keystonerc_admin**, so we will use this.

The process of *logging in* to the admin user then becomes:

```
$ source /root/keystonerc_admin
```

Listing A.16: Logging in as admin

A.11 Creating the external network

For this network there are a few pre-requisites. The network in use needs to have a range of ip address available to OpenStack that won't be allocated by something else through DHCP or other means. On the network where this was set up the addresses from *10.0.0.170* to *10.0.0.179* were available, so these will be used. The gateway is at *10.0.0.1* and the subnet is 10.0.0.0/24.

The first thing that needs to be done is to create an external net within OpenStack. Then we need to create a subnet within the network that contains the routing information for our network. The following commands does this, but the network information will likely be different for someone else setting this up.

```
$ neutron net-create ext-net --router:external True --provider:physical_network external
    --provider:network_type flat
$ neutron subnet-create ext-net --allocation-pool start=10.0.0.170,end=10.0.0.179 --
    gateway 10.0.0.1 --enable_dhcp=False --name ext-subnet 10.0.0.0/24
```

Listing A.17: Setting up the external network

Note down the id of the network, it will be refered to later as **ext-net-id**.

In addition to creating the networks we need to create a router. Create it using the following command and note down the id which will be refered to later as **router1-id**

```
$ neutron router-create router1
```

Listing A.18: Creating the router router1

The router then needs to be connected together with ext-net. This is done by the following command, but **ext-net-id** and **router1-id** needs to be replaced by the ids noted down earlier.

```
$ neutron router-gateway-set router1-id ext-net-id
```

Listing A.19: Connecting the router to ext-net

A.12 Creating the first user

The network is up and running, but we need to create a user and give that user access to use the network.

Note down the id returned by the first command and use it as the tenant-id in the second one.

```
$ keystone tenant-create --name <username>
$ keystone user-create --name <username> --tenant-id <tenant-id> --pass <password> --
    enabled true
```

Listing A.20: Creating the first user

Copy the keystonerc_admin file to keystonerc_<username>, update all instances of *admin* to <*username*> and update the password to the one that was set in the above command.

We need to set a role to the user, which acts as a set of permissions - for simplicitys sake the new user is going to be an admin user with all permissions.

```
$ keystone user-role-add --role admin --tenant <tenant-id> --user <user-id>
```

Listing A.21: Making the user admin

If the tenant-id and user-ids weren't noted down they can be found with the following commands.

```
$ keystone tenant-list
$ keystone user-list
```

Listing A.22: Listing the tenant-id and user-id

The user is going to have the role **heat_stack_owner** in order to be able to interact with **Heat**. This is done the same way as admin was added to the user.

```
$ keystone user-role-add --role heat_stack_owner --tenant <tenant-id> --user <user-id>
```

Listing A.23: Making the user heat_stack_owner

A.13 Creating the user-net

Log into the new user and create a network with an ip range. The ip range will only be used internally on the OpenStack systems, so just pick something that won't be in use.

```
$ source keystonerc_<username>
$ neutron net-create UserNet
$ neutron subnet-create UserNet 192.168.90.0/24
```

Listing A.24: Creating the network for the user

Log back into the admin user and add the subnet as an interface on the router created earlier.

```
$ source keystonerc_admin
$ neutron router-list
$ neutron subnet-list
$ neutron router-interface-add <router-id> <user-subnet-id>
```

Listing A.25: Adding the users net as an interface to the router

Log back into the user.

```
$ source keystonerc_<username>
```

A.14 Creating the first VM

To make sure everything is working a cloud image will be downloaded and started. First install wget and download an image - for this guide I selected **Fedora**.

```
$ yum install -y wget
$ wget https://download.fedoraproject.org/pub/fedora/linux/releases/24/CloudImages/x86_64
    /images/Fedora-Cloud-Base-24-1.2.x86_64.qcow2
```

Listing A.26: Installing wget and downloading Fedora

OpenStack has a detailed list of images available in the documentation[44].

Glance is the OpenStack module that stores objects like disk images. The Fedora image needs to be imported into Glance before use. We also need to create an ssh-key to ssh into the cloud image after it is started. The following commands perform those actions.

```
$ glance image-create --container-format=bare --disk-format=qcow2 --name=fedora24 <
    Fedora-Cloud-Base-24-1.2.x86_64.qcow2
$ nova keypair-add testkey > testkey.key
```

Listing A.27: Importing Fedora image into Glance and creating an ssh-key

By default the security rules don't allow any traffic to the VMs - these are edited to allow ICMP (ping) and ssh traffic. Since the user that was created is admin there will be two default security groups. To list the security groups:

```
$ neutron security-group-list
```

Listing A.28: Listing the security groups

Rules are added to both the groups:

```
$ neutron security-group-rule-create --protocol icmp --direction ingress <security-group-
    id-1>
$ neutron security-group-rule-create --protocol icmp --direction ingress <security-group-
    id-2>
$ neutron security-group-rule-create --protocol tcp --port-range-min 22 --port-range-max
    22 --direction ingress <security-group-id-1>
$ neutron security-group-rule-create --protocol tcp --port-range-min 22 --port-range-max
    22 --direction ingress <security-group-id-2>
```

Listing A.29: Allowing ICMP and ssh

A.15 Booting the VM

To boot the VM we use nova. There are several preset *flavors* of hardware available, which means a set of memory, processors and disk space. We will chose the flavor **2** which is the smallest that is able to run fedora.

```
$ nova boot --flavor 2 --image fedora24 --key-name testkey <VM name>
$ nova list
```

Listing A.30: Starting the VM

The last command is repeated until the VM is *Running*. Note down the VMs ID.

Find the id of the VMs network port by doing:

```
$ neutron port-list --device_id <VM id>
```

Listing A.31: Finding the VMs network port id

Now create a floating ip (external ip) and associate it with the VMs network port.

```
$ neutron floatingip-create ext-net
$ neutron floatingip-associate <floatingip-id> <VMs port-id>
```

Listing A.32: Creating floatingip and associating it with the VM

You should now see the external ip when you execute **nova list** and the VM should be accessible from the outside. SSH into the vm to verify that everything works.

```
$ ssh fedora@<ip> -i testkey.key
```

Listing A.33: SSHing into the vm

A.16 Setting up DNS

The virtual machines are still not able to resolve DNS querries, for this we need to add a DNS server to the information the VM receives.

First find the id of the subnet belonging to the user, and not the external net, which is the one that was created in A.24.

```
$ neutron subnet-list
```

Listing A.34: Finding the id of the user net

Add at least two DNS servers to this subnet - on the lab I am working on there is only one DNS server - **10.0.0.30** and no outside DNS traffic allowed, so the second DNS server I added was just the loopback device (**127.0.0.1**). If you do not have such limitations on your network you may use the DNS servers operated by Google (**8.8.8.8, 8.8.4.4**).

```
neutron subnet-update 8331ccf5-d74c-413d-b920-91054c89ee5a --dns_nameservers 10.0.0.30
    127.0.0.1
```

Listing A.35: Adding DNS servers

These sources[44][45][46][47][48][49][50] were used for the installation process.

Appendix B

Failed OpenStack installation attempt

B.1 Installation using puppetlabs-openstack

The initial idea for the project was to use the configuration management tool Puppet to set up the OpenStack and ELK environments. The ELK stack has since been removed from the project due to time limitations and narrowing of the thesis scope.

There was already a Puppet module available for installing OpenStack named **puppetlabs-openstack** which is linked from the official OpenStack website, so it was used for this project.

The Puppet version in use ran on a version of Ruby which was more recent than the one expected by by the Puppet module, and as such did not automatically include the **rubygems** package. A result of not having rubygems was that none of the requirements worked, so the environment variable **RUBYOPT** was set to *"rubygems"* to include it globally and fix the issue.

The Puppet function **file_concat** exists in some versions of Puppet, but not the one in use by this project. The fix for this was to simply install the Puppet module **electrical-file_concat** which provides the functionality and should possibly be concidered a dependency for the **puppetlabs-openstack** module.

Another issue was that the **puppetlabs-openstack** module depends on the **puppetlabs-mysql** module which depends on either the package **mysql** and a service named **mysqld** or the package **mariadb** and the service named **mariadb**. Mariadb is a binary replacement for mysql that has been rapidly gaining traction as many distributions have replaced the mysql package with mariadb. CentOS has replaced the mysql package with mariadb, but kept the mysql service name to prevent old scripts form needing changes, but for some reason the **puppetlabs-mysql** module has a section hard-coded for CentOS with the wrong values for both the provider (mysql instead of mariadb) and service name (mariadb instead of mysqld). Updating the file */etc/puppet/modules/mysql/manifests/params.pp* with the correct values allows both mysql and OpenStack to install.

After the install VMs would start, but the network did not work. Some changes were made to the network, and after a reboot the server did not connect to the network again. At this point it was decided that a reinstall would be the easiest solution in addition to verifying that the now-modified Puppet modules works and that the system could actually be rebuild using only them. The reinstall was made, but at some point in time since OpenStack was originally installed with Puppet the version of OpenStack that was in use got the status EOL (End-Of-Life) and got removed from the software repositories, preventing further installation.

B.2 Conclusion

The latest version of the **puppetlabs-openstack** module points to a version of OpenStack that is EOLed, and requires lots of hotfixes to work in the first place, which makes me conclude that this way of installing OpenStack is simply not mature enough at the time of writing. OpenStack is a fast moving target where two new versions are released each year and each version is only considered stable for 6 months and the Puppet OpenStack module has simply not been able to keep up with the pace.

For these reasons I conclude that using these Puppet module is not a good way to install OpenStack.

Bibliography

[1] Ian Foster. what is the grid? a three point checklist. *Grid Today*, 1(6), 2002. https://www.mcs.anl.gov/~itf/Articles/WhatIsTheGrid.pdf.

[2] Ian Foster and Carl Kesselman. *The grid: blueprint for a new computing infrastructure*. Morgan Kaufmann Publishers Inc, 1999.

[3] Parvin Asadzadeh et al. Global grids and software toolkits: A study of four grid middleware technologies. Technical report, University of Melbourne, Australia, July 2004. Url: https://arxiv.org/abs/cs/0407001.

[4] Alice environment grid framework. http://alien.web.cern.ch/.

[5] condor submit manual. https://research.cs.wisc.edu/htcondor/manual/current/condor_submit.html.

[6] The large hadron collider. http://home.cern/topics/large-hadron-collider.

[7] V Welch et al. Globus toolkit version 4 grid security infrastructure: A standards perspective. *The Globus Alliance*, 2005.

[8] Proxy certificate profile. https://www.ietf.org/rfc/rfc3820.txt.

[9] Peter Mell and Timothy Grance. The nist definition of cloud computing. http://dx.doi.org/10.6028/NIST.SP.800-145, 2011.

[10] ALICE-Tier-1/Tier-2-Workshop. Alice-tier-1/tier-2-workshop. https://indico.cern.ch/event/485835/, 2016.

[11] Companies supporting the openstack foundation. https://www.openstack.org/foundation/companies/.

[12] Nova's developer documentation. https://docs.openstack.org/developer/nova/.

[13] Openstack block storage cinder. https://wiki.openstack.org/wiki/Cinder.

[14] Paul Rad, Rajendra V Boppana, Palden Lama, Gilad Berman, and Mo Jamshidi. Low-latency software defined network for high performance clouds. In *System of Systems Engineering Conference (SoSE), 2015 10th*, pages 486–491. IEEE, 2015.

[15] Senlin overview. https://wiki.openstack.org/wiki/Senlin.

[16] First commit to senlin repository. https://git.openstack.org/cgit/openstack/senlin/commit/?id=4732854d10fef68f32297a772c550cf31c4e964a.

[17] Yoji Yamato, Masahito Muroi, Kentaro Tanaka, and Mitsutomo Uchimura. Development of template management technology for easy deployment of virtual resources on openstack. *Journal of Cloud Computing 3.1*, 3(1), 2014.

[18] Wikipedia: Orchestration (computing). https://en.wikipedia.org/wiki/Orchestration_(computing).

[19] Lars Kellogg-Stedman. Figure from presentation. http://oddbit.com/rdo-hangout-heat-intro/#/8.

[20] Heat orchestration template (hot) specification: Resources section. http://docs.openstack.org/developer/heat/template_guide/hot_spec.html#resources-section.

[21] Ceilometer developer documentation. https://docs.openstack.org/developer/ceilometer/.

[22] Openstack ceilometer and the gnocchi experiment. https://julien.danjou.info/blog/2014/openstack-ceilometer-the-gnocchi-experiment.

[23] Openstack ceilometer and the gnocchi experiment. https://julien.danjou.info/blog/2014/openstack-ceilometer-the-gnocchi-experiment.

[24] Openstack telemetry. https://wiki.openstack.org/wiki/Telemetry#Aodh.

[25] Intel vt vs. amd pacifica. https://web.archive.org/web/20130530214041/http://www.informationweek.com/intel-vt-vs-amd-pacifica/172302134.

[26] Joanna Rutkowska and Rafal Wojtczuk. Qubes os architecture. *Invisible Things Lab Tech Rep*, 2010.

[27] Bjarte Kileng and Boris Wagner. Dynamic virtualization tools for running alice grid jobs on the nordic arc grid. *NIK 2013*, pages 158–161, 2013.

[28] P Buncic, C Aguado Sanchez, J Blomer, L Franco, A Harutyunian, P Mato, and Y Yao. Cernvm – a virtual software appliance for lhc applications. *Journal of Physics: Conference Series*, 219(4):042003, 2010.

[29] Why make scientific linux. https://www.scientificlinux.org/about/why-make-scientific-linux/.

[30] Dario Berzano. elastiq readme.md. https://github.com/dberzano/elastiq, 2014.

[31] Joachim Christoffer Carlsen. Elastic grid resources using cloud technologies. Master's thesis, Bergen University College, University of Bergen, 2014.

[32] Aliec2 repository. https://github.com/Joachricar/AliEC2.

[33] Experience with dynamically provisioned worker nodes at the ral tier-1. https://indico.cern.ch/event/247864/contributions/1570372/attachments/426730/592316/AndrewLahiff_RALVMs_HEPiX2013.pdf.

[34] Devstack documentation. https://docs.openstack.org/developer/devstack/.

[35] Anvil documentation. https://anvil.readthedocs.io/en/latest/.

[36] Openstack deployment | installer | fuel. https://www.mirantis.com/software/openstack/fuel/.

[37] Puppetlabs openstack github repository. https://github.com/puppetlabs/puppetlabs-openstack.

[38] Anshu Awasthi and P Ravi Gupta. Comparison of openstack installers. *International Journal of Innovative Science, Engineering & Technology*, 2(9), 2015.

[39] T Bell, B Bompastor, S Bukowiec, J Castro Leon, M K Denis, J van Eldik, M Fermin Lobo, L Fernandez Alvarez, D Fernandez Rodriguez, A Marino, B Moreira, B Noel, T Oulevey, W Takase, A Wiebalck, and S Zilli. Scaling the cern openstack cloud. *Journal of Physics: Conference Series*, 664(2), 2015.

[40] The OpenStack Foundation. Networking with nova-network. http://docs.openstack.org/admin-guide/compute-networking-nova.html.

[41] The OpenStack Foundation. Openstack releases. https://releases.openstack.org/.

[42] packstack fails to configure ovs bridge for centos. https://bugzilla.redhat.com/show_bug.cgi?id=1316856.

[43] auto-scaling cooldown fails to adjust. https://bugs.launchpad.net/heat/+bug/1569273.

[44] The OpenStack Foundation. Get images. http://docs.openstack.org/image-guide/obtain-images.html, 2016.

[45] The RDO Project. Neutron with existing external network. https://www.rdoproject.org/networking/neutron-with-existing-external-network/, 2016.

[46] The RDO Project. Packstack all-in-one diy configuration. https://www.rdoproject.org/documentation/packstack-all-in-one-diy-configuration/, 2016.

[47] The OpenStack Foundation. External network. http://docs.openstack.org/juno/install-guide/install/apt/content/neutron_initial-external-network.html, 2016.

[48] Red Hat Inc. Interface configuration files. https://www.centos.org/docs/5/html/Deployment_Guide-en-US/s1-networkscripts-interfaces.html, 2016.

[49] The RDO project. Packstack quickstart: Proof of concept for single node. https://www.rdoproject.org/install/quickstart/, 2016.

[50] The OpenStack Foundation. Debug dns issues. http://docs.ocselected.org/openstack-manuals/kilo/networking-guide/content/debugging_dns_issues.html.