KTH Royal Institute of Technology

Master of Science Thesis

KTHFS Orchestration – PaaS orchestration for Hadoop

Author:	Alberto Lorente Leal
Examiner:	Prof. Seif Haridi
Supervisor:	Dr. Jim Dowling
Programme:	Software Engineering of Distributed Systems

TRITA-ICT-EX-2013:175

Abstract:

Platform as a Service (PaaS) has produced a huge impact on how we can offer easy and scalable software that adapts to the needs of the users. This has allowed the possibility of systems being capable to easily configure themselves upon the demand of the customers. Based on these features, a large interest has emerged to try and offer virtualized Hadoop solutions based on Infrastructure as a Service (IaaS) architectures in order to easily deploy completely functional Hadoop clusters in platforms like Amazon EC2 or OpenStack.

Throughout the thesis work, it was studied the possibility of enhancing the capabilities of KTHFS, a modified Hadoop platform in development; to allow automatic configuration of a whole functional cluster on IaaS platforms. In order to achieve this, we will study different proposals of similar PaaS platforms from companies like VMWare or Amazon EC2 and analyze existing node orchestration techniques to configure nodes in cloud providers like Amazon or Openstack and later on automatize this process.

This will be the starting point for this work, which will lead to the development of our own orchestration language for KTHFS and two artifacts (i) a simple Web Portal to launch the KTHFS Dashboard in the supported IaaS platforms, (ii) an integrated component in the Dashboard in charge of analyzing a cluster definition file, and initializing the configuration and deployment of a cluster using Chef.

Lastly, we discover new issues related to scalability and performance when integrating the new components to the Dashboard. This will force us to analyze solutions in order to optimize the performance of our deployment architecture. This will allow us to reduce the deployment time by introducing a few modifications in the architecture.

Finally, we will conclude with some few words about the on-going and future work.

Key words: *KTHFS, HDFS Orchestration, Chef, Jclouds, Amazon EC2, Openstack, PaaS.*

Acknowledgements:

First, I would like to thank Dr. Jim Dowling for his support and kindness during this work. Without his aid, valuable vision and interest on cloud computing I would not have been able to reach so far in this work, plus also thank for the hours and support he gave during the development of this thesis, including the opportunity to be part of this project at SICS. In addition, I would like to thank the professors of KTH Royal Institute of Technology for developing myself in this big area of expertise in distributed systems.

Secondly I would like to thank the researchers at SICS. It was inspiring and motivating to work in such a great environment with so many people from different places and cultures. Most importantly, thank the people involved in the KTHFS project and the possibility to work side-by-side with such exceptional partners in such a small team for these last months working together.

On the other hand, I would like to thank my friends from the IEEE Student Branch from the ETSIT-UPM for their support and for the years in Madrid where I was able to develop my unconditional love to the things I like to do when it comes to Information Technology and the challenges it presents, plus the possibility to improve myself as a better person and professional.

Lastly and more importantly, I would like to thank my family, my parents and my brother for their unconditional love and support both financially and encouragement during my studies both in Madrid and in Stockholm.

June 24, Stockholm, Sweden

Table of Contents

Table of Figures

Table of Tables

Chapter 1- Introduction:

In these last few years, Cloud Computing and Big Data are experiencing a high peak of demand where storing large amounts of data is becoming a key crucial aspect for companies in order to offer better services to their customers. There is a race competition with the release of crucial updated systems which try to offer high availability in error prone scenarios where nodes are capable of failing unexpectedly. This is the case of the prototype system KTHFS [1] [2], a customized version of HDFS [3] which tries to deliver a highly available and scalable system addressing the limitations of the original Hadoop platform.

With the actual enhancements KTHFS offers, its architecture allows it to achieve higher scalability and availability of the Hadoop namenode. Nowadays, there has been interest of bringing together PaaS and HDFS due to the new challenges that it offers and its opportunities that we will discuss later. Therefore, there is a high interest to extend the capabilities of KTHFS in order to offer PaaS [4] capabilities on top of existing IaaS [4] architectures like Amazon Web Services [5] or Openstack [6]. This also provides the possibility for the system to conform to the needs of the platform and adapt upon failures or extend its computing power. More importantly, it will facilitate the usage of the platform, as it will drastically reduce the amount of time and resources needed to configure and deploy the platform.

To achieve this, we will study similar approaches that are currently in development and their proposed solutions, we will discuss them in the following sections. Moreover, we will analyze the different technologies which surround the PaaS provisioning ecosystem which will help us define how we can achieve PaaS capabilities in our platform. This will include the study of existing orchestration languages and define the guidelines of our own language which will match our needs for deploying KTHFS.

1.1 What is Cloud Computing?

The notion of high performance computing is suffering a radical change these days. In the old days of Moore's Law [7], people remarked how the power of the transistor in modern computers will allow more performance by the increase in processing power of modern computers. In order to continue maintaining the future of Moore's Law, a new solution had to be elaborated as we reached the limits of the electric properties of silicon and we could not scale more by increasing the frequency rate of the transistors. This leads to what we know now as multi core architectures where instead of increasing frequency power, the number of transistors is increased by allocating more CPUs in the same chip.

Nowadays, as multi core architectures become more powerful; new paradigms appear in order to achieve better scalable and high performing applications by using the processing power in a single cluster or by allocating a large number of clusters in huge data centers.

We can define cloud computing as the next phase to attain high performance computing, by allocating more computing power and resources by managing large clusters of nodes which working together offers a high processing power. Moreover these systems need to be autonomous in the sense that they need to adapt to peaks of high demand and allocate more resources to sustain this demand. It has also become a new way of doing business where large companies like Amazon can offer the computing power and elasticity of their data centers and bill proportionally to their resources usage [5].

NIST defines cloud computing as the following:

"Cloud computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction [4]."

1.2 IaaS, PaaS and SaaS:

With the emergence and evolution of cloud based services like Amazon Web Service, different service models have been defined when talking of service oriented business models in cloud computing [4][8]:

1.2.1 Traditional IT Stack

In the traditional IT model, software development will be carried along the guidelines of the requisites of the client to achieve the desired product. This will produce software adapted to the client's immediate needs. It allows the highest level of customization to the client as the software is developed as specifically tailored for the customer's needs.

1.2.2 Infrastructure as a Service (IaaS)

In this provisioning model, the provider manages and controls the whole infrastructure from the software stack. This usually of the underlying layers related to networking, virtualization, storage and server HW. This normally is translated that the provider's Infrastructure manages the deployment of virtual machines which later on, the user will be able to configure and customize the contents of these machines to achieve their business needs. Virtually, the service provider handles the infrastructure resources so that the user can think on how to use the available hardware resources for their needs. In this case the billing model is based on the level of virtual machines and resources allocated for their needs.

1.2.3 Platform as a Service (PaaS)

In this service model, the service provider charges the users for the usage of their platform. This is translated to the fact that the service provider offers his architecture to clients where they can deploy their applications automatically without the need of the client to do any configuration by its side. This means that the user has the freedom to customize their applications for their needs but the rest of the underlying layers are indirectly handled by the platform offered by the service provider. In contrast to IaaS, extra guarantees are offered in this kind of services, like for example elasticity properties on their applications. This enables applications to scale dynamically and allocate more resources automatically to satisfy the peak of demand as it takes place [9].

1.2.3 Software as a Service (SaaS)

It was initially thought as a business model with the design and implementation of service oriented software architectures. It is a business model in which the user does not own the software but pays for its usage. In this sense compared to PaaS, the user is not able to use their applications as they are enforced to use the applications offered by the vendor. The service provider in this case handles the whole stack, managing every detail of the software service (hosting, management, security) and offering their software solution to the client by deploying its service on the Internet.

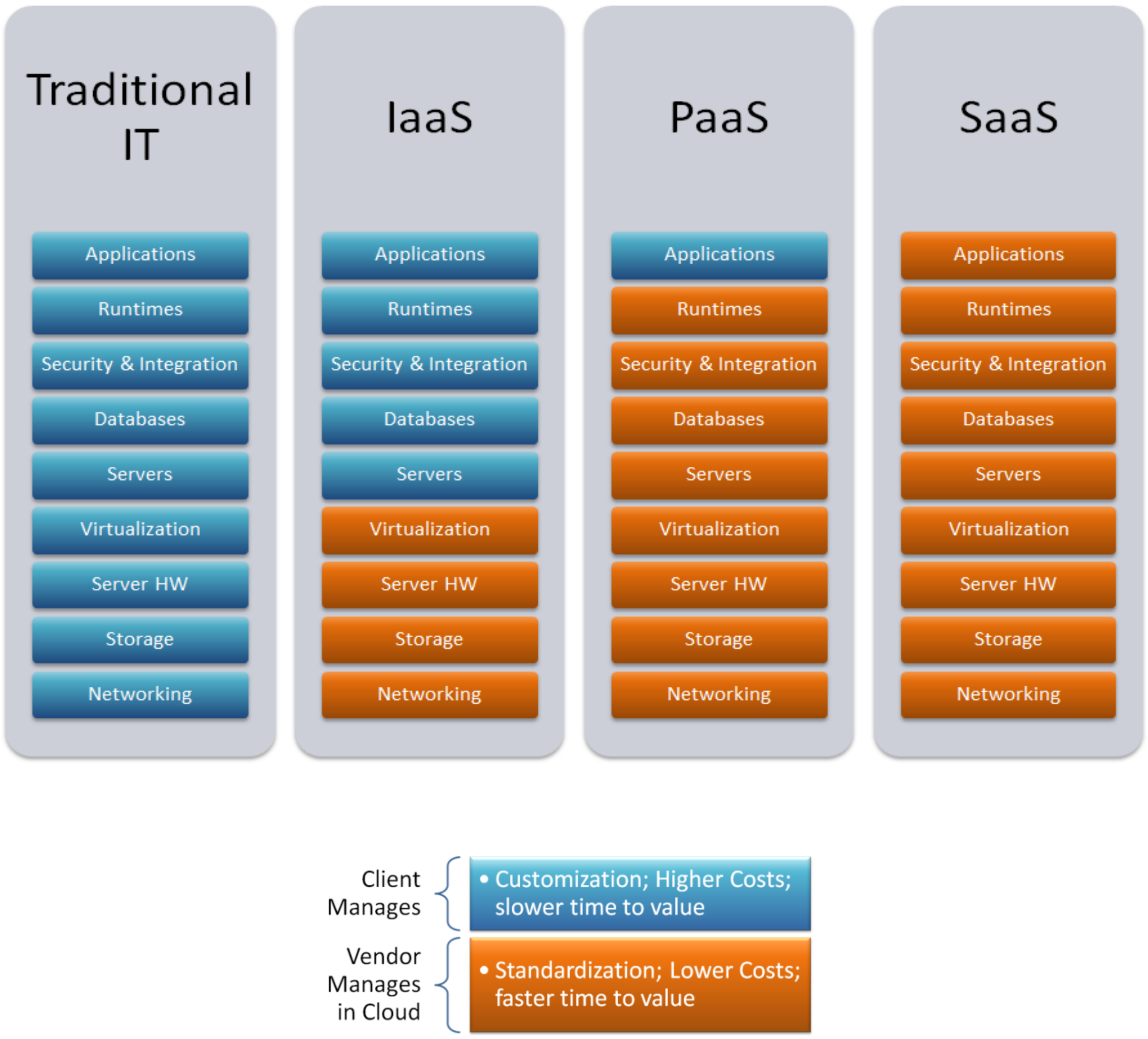


Figure 1: Structure of software services

Chapter 2 - Background:

In this chapter, we will elaborate a brief overview for our interest to propose a solution for our problem considering different proposals from other works in progress. Firstly, we will give an overview of the different next generation Hadoop platform been designed until now and we will also present very briefly KTHFS [1] [2] a distributed file system been developed in SICS and where we will focus our efforts on. Secondly, we will give an overview of PaaS solutions that has been used to deploy applications and services over IaaS solutions. Finally, we will introduce two IaaS platforms that our solution will be design towards for. In addition, we will motivate the reasoning behind the development of virtualized solutions for Hadoop.

2.1 Why Virtualize Hadoop?

One of the motivations behind this thesis work was to try to bring closer together PaaS and the world of HDFS. This has led to the interest of trying to offer HDFS solutions that can be easily deployed in cloud environments. What are the advantages of migrating from physical machines to virtualized? What are the disadvantages? Is it worth spending all this time and effort to migrate Hadoop on to cloud environments? In this section, we emphasize aspects for using virtualized environment and will do briefly based on descriptions of scenarios [10] and previous analysis from companies like VMware with Project Serengeti [11].

2.1.1 Advantages:

- **Hardware:** Virtual machines follow specified configuration specifications and multiple profiles can be chosen. This approach avoids possible inconsistencies of using multiple machines with different hardware configurations as we can allocate multiple virtual machines under the same configuration profile.

- **Rapid Provisioning:** We can create virtual machine images that can act as templates in order to have pre-configured images with all the software needed to configure each of the software services which compose a Hadoop cluster.

- **Cluster Isolation:** Achieve improved security isolation of the Hadoop nodes with the help of cloud infrastructures. We can isolate groups of nodes through the definition of security groups, offering data isolation while running the nodes in the same machine.

- **Networking:** No need to manually configure the network that underlies the cluster, all the virtual machines allocate IP addresses automatically and configure the necessary ports.

- **Elasticity:** One attractive property that PaaS offers the possibility of dynamically provides extra resources when the system detects that the application is reaching a peak of demand. In the case of Hadoop, this means that we can provide a Hadoop cluster that dynamically will allocate new Hadoop nodes automatically without the need of interacting with the system.

2.1.2 Disadvantages:

- **Topologies:** Normally Hadoop is provisioned in scenarios where we are considering static topologies. In cloud infrastructures we need to work with the notion of dynamic topologies, where nodes may come in or leave unexpectedly.

- **Static Infrastructures:** Hadoop is assumed to work in static environments. This is translated to how scheduling and recovery algorithms work for normal Hadoop environments. Therefore, we need to take care of how the Hadoop recovery algorithms will react when working in a dynamic virtualized environment.

- **Failures and Data:** Hadoop assumes failures to occur independently and allocate data upon this assumption. In the case of cloud infrastructures, this is not the case as multiple nodes could be allocated in the same zone or region assuming that if a node failures, probably others will do.

2.2 Virtualized Hadoop Platforms:

Virtualized Hadoop is the effort of managing and deploying Hadoop clusters by using Hadoop nodes that are allocated by virtual machines in a virtual environment. As we previously explained, it offers a series of interesting properties that may benefit Hadoop and so, in this section we will look deeply into two actual solutions which are available for its usage and also introduce KTHFS.

2.2.1 VMware Serengeti:

Overview:

VMware's Project Serengeti [12] is an effort to offer a highly available and virtualized HDFS infrastructure. It uses VMware's vSphere [13] to obtain a highly available Hadoop platform allowing the possibility of deploying different versions of HDFS. It has also been tagged as an open source project.

Architecture:

In VMware's solution, the virtualization layer of the platform is handled by VMware's vSphere which monitors and restarts the VMs (Virtual Machines) in case of detecting a failure. In this system, they include a series of predefined virtual machines, which cover key components like the Serengeti Management Server and a Hadoop Template Virtual Machine, a predefined VM image to help provisioning of HDFS nodes in the virtual environment [14]. One of the key components in their solution is the definition of an agent which is responsible for configuring, downloading and installing the desired Hadoop software [14].

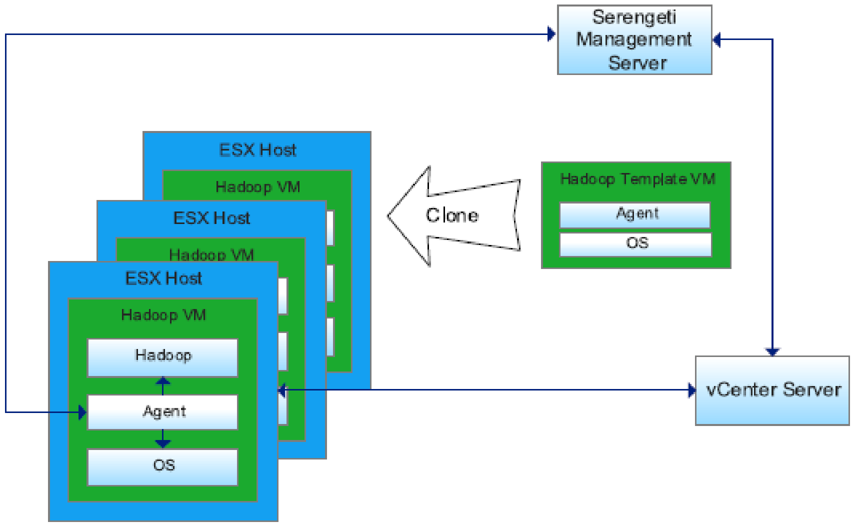


Figure 2: Virtual machine provision

In this case, the provisioning of the platform is achieved through the definition of configuration parameters in the manifest file accessible through the Serengeti's source code. In this file, we can define the distributions that we want Serengeti to have available [14]. The definition of Hadoop service roles and their related software is defined through a JavaScript Object Notation (JSON) file. JSON [15] is an open standard which tries to offer a human readable format for serializing and transmitting data. In the case of Serengeti, it has the following structure:

```
  {
    "name" : "mapr",
    "vendor" : "MAPR",
    "version" : "2.1.1",
    "packages" : [
      {        "roles" : ["mapr_zookeeper", "mapr_cldb", "mapr_jobtracker", "mapr_tasktracker",
"mapr_fileserver", "mapr_nfs", "mapr_webserver", "mapr_metrics", "mapr_client", "mapr_pig",
"mapr_hive", "mapr_hive_server", "mapr_mysql_server"],
        "package_repos" : ["http://<ip_of_serengeti_server>/mapr/2/mapr-m5.repo"]
      }
    ]
  },
  {
    "name" : "cdh4",
    "vendor" : "CDH",
    "version" : "4.1.2",
    "packages" : [
      {        "roles" : ["hadoop_namenode", "hadoop_jobtracker", "hadoop_tasktracker",
"hadoop_datanode", "hadoop_journalnode", "hadoop_client", "hive", "hive_server", "pig",
"hbase_master", "hbase_regionserver", "hbase_client", "zookeeper"],
        "package_repos" : ["http://<ip_of_serengeti_server>/cdh/4/cloudera-cdh4.repo"]
      }
    ]
  }
```

Figure 3: Sample of services definition

On the other hand, by reusing the previous definitions; we can deploy a cluster using a JSON defining the structure of our desired HDFS cluster; a sample is as follows [14]:

```
{
"nodeGroups" : [
                    {
                       "name": "master",
                       "roles": [
                       "hadoop_namenode",
                       "hadoop_jobtracker"
                        ],
                       "instanceNum": 1,
                       "instanceType": "MEDIUM"
                     },
                    {
                        "name": "worker",
                        "roles": [
                        "hadoop_datanode", "hadoop_tasktracker"
                         ],
                        "instanceNum": 5,
                        "instanceType": "SMALL"
                     },
                    {
                       "name": "client",
                       "roles": [
                       "hadoop_client",
                       "hive",
                       "hive_server",
                       "pig"
                        ],
                       "instanceNum": 1,
                       "instanceType": "SMALL"
                     }
                     ]
}
```

Figure 4: Cluster definition sample VMware Serengeti

Following this previous structure, a user can define in this case the group of nodes that compose the cluster. Additionally, in each group we can specify the number of nodes we want the system to deploy, the type of virtual instance we want to use and the software services to be installed in each machine.

2.2.2 Hortonworks Data Platform:

Overview:

Hortonworks is a reseller which focuses in offering solutions when deploying and configuring Hadoop. They offer a Hadoop infrastructure planned to scale and be deployed in virtual environments or in physical machines. It is designed to provision a Highly Available and Scalable system with ease of use and without any extra costs by offering a pre-integrated package with all the essential Apache Hadoop Components (HDFS, MapReduce, Pig, HBase and ZooKeeper)[16].

Architecture:

In this case, Hortonworks proposal is centered in offering both a predefined Hadoop distribution and a sandbox solution so that their user may deploy and test their software in their clusters. One of the key features in this case is that their architecture offers a highly available Hadoop platform [17]. It also offers a web application that allows a cluster management allowing the user to provision and monitor their Hadoop cluster. This application is called Apache Ambari and it is available as an open source project.

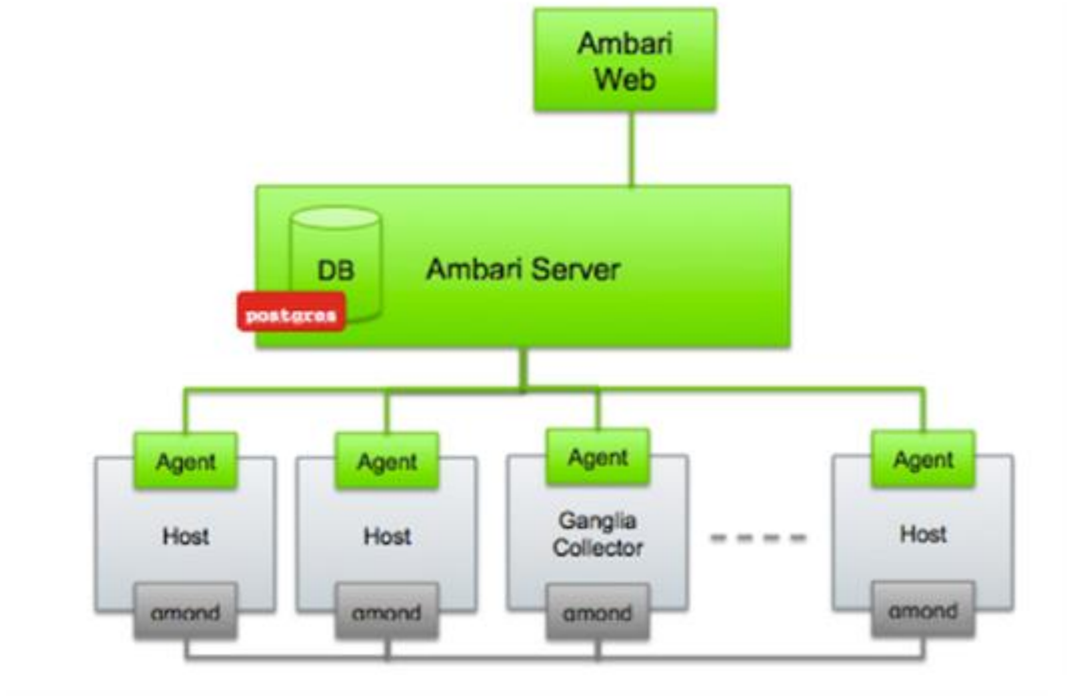


Figure 5: Ambari architecture

In this case, the architecture works as follows. The user using the Ambari web, will provide the SSH key of the hosts and a list of hosts to connect. With this approach, it can work with virtual or physical machines as it only requires the hosts address. Then, the application will connect to the machines and install the Ambari agents that will provide information of these machines. After that, using Ambari web; the user will select the services to be configured in those machines by sending commands to the Ambari agents stored in the machines. It makes use of Puppet [18] to deploy and control the nodes configuration and Ganglia [19], Nagios [20] to monitor and alert of the status of the cluster nodes [21]. Puppet is a configuration management platform that allows users to write small programs in ruby that will be in charge of installing the software packages and binaries that we want in the machine.

2.2.3 KTHFS:

Overview:

It is a distributed file system that is currently under development in SICS. The development and enhancements has been carried by previous Master Thesis students that addressed some of the limitations of the system [1] [2]. This system is based on the original Hadoop source code from the Apache foundation and some of it key features compared to the original Hadoop file system are that it offers a highly available and more scalable HDFS.

Architecture:

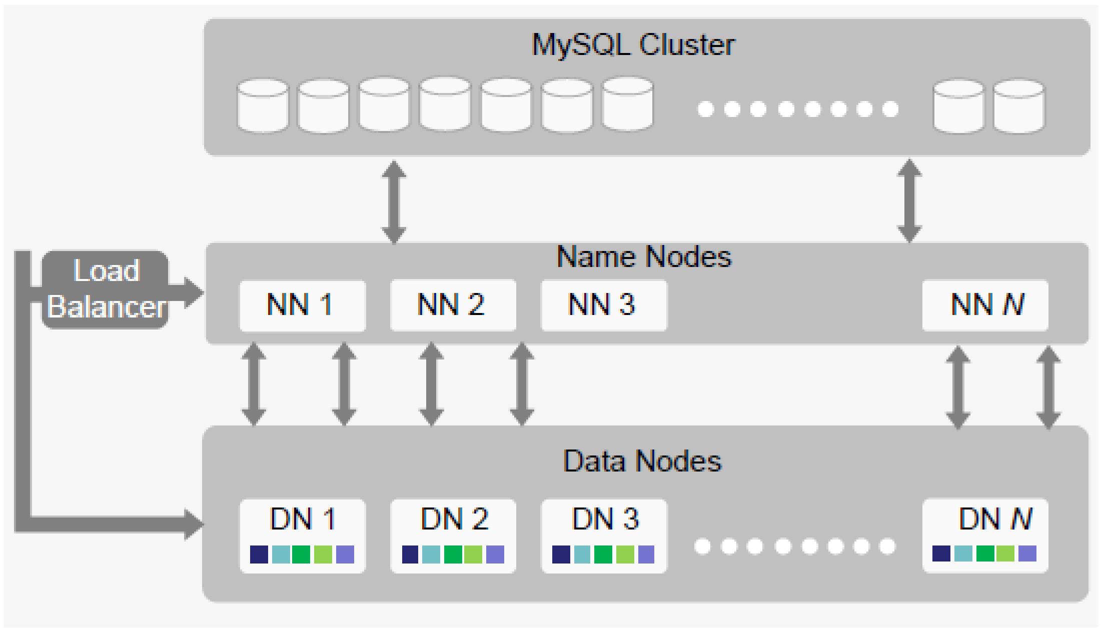


Figure 6: KTHFS architecture

In this case, the KTHFS architecture addresses the limitations the original HDFS had regarding the scalability and availability of the name node by:

1. Replacing the original HDFS name node by multiple stateless name nodes obtaining high availability. In this case if the master name node crashes, another name node can take its place.
2. Moving the metadata stored in the original name node into an underlying layer composed by a MySQL Cluster [22]. This way, we can store more metadata compared to the original HDFS.

One of the components that will be of our interest in this case, will be the KTHFS Dashboard; a web application that monitors and retrieves information about the status of the nodes that compose the KTHFS cluster. In our case, we will focus to enhance its functionality in order to make it functional not only on physical environments but virtualized cloud environments. We will discuss this during the following chapters that compose this thesis.

2.3 IaaS Platforms:

2.3.1 Amazon Web Services:

The Amazon EC2 infrastructure [5] allows the possibility of users with an Amazon Web Services account to provision and deploy virtual machines in their public cloud. They offer the possibility to launch nodes and select VMs through their EC2 console. They offer a large selection of Amazon Machine Images (AMIs) which can be selected by the users by prior payment or in some cases for free. The platform is designed to satisfy a large variety of flavours to configure the virtual machines when it comes to selecting the architecture and resources for our application. A public API is available on their development center so users can programmatically set up their virtual machines without the need to interact directly by login through the EC2 Management Console. It is one of the most widely and used platforms when it comes to cloud services [23].

2.3.2 OpenStack:

OpenStack [6] is an open source project managed by the OpenStack foundation with the aim of offering an open design and development process guidelines for Infrastructure as a Service. It aims to offer an open standard so clients can develop their own cloud data centers easily over standard hardware solutions in order to scale their platforms. New modules are proposed by the community, which are referred to be in an incubation state; during this phase the main characteristics of this module are defined taking a process of six months at least [24]. After the incubation phase, the new module is added to form part of the technology stack of the OpenStack framework. This way new feature are designed and included for the whole community. The key components that form foundation of the Openstack are in this case Nova and Swift. The Nova component handles the compute service, in charge of deploying virtual instances and resources; while the Swift component handles storage services so users can specify storage to be attached on the virtual instances.

2.4 PaaS Solutions:

2.4.1 InfoChimps Ironfan:

Infochimps is a big data company focused in developing tools and technologies for provisioning of services and big data clusters in the cloud. In this case, their main work focuses in a tool called Ironfan [25] which is an orchestration Domain Specific Language (DSL). This tool works on top of Chef [26] to allows clients to define their clusters in a Ruby file which is interpreted by Ironfan. After the interpretation phase, it launches and configures the specified virtual machines with the desired software using the user credentials in the cloud of their choice. For now it comes with full support for Amazon EC2 and they also plan future support for OpenStack based platforms. The remarkable thing with their solution is that it includes other features, like service discovery which allows the nodes to identify the rest of the machines involved in the cluster and automatically interconnect the services between them.

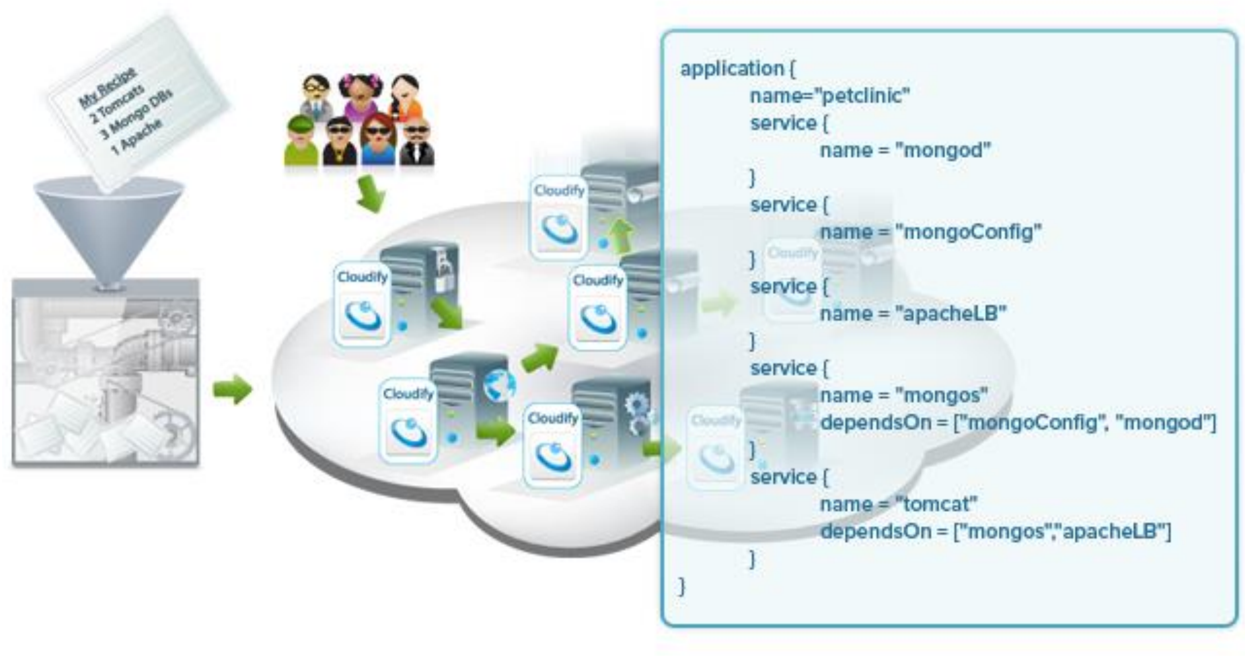
2.4.2 Amazon OpsWorks:

Amazon being one of the big companies when it comes to offering IaaS solutions recently launched their own proposal to offer PaaS in their Amazon EC2 platform. This new service is called Amazon OpsWorks [27]. With this service, the clients can freely handle and manage their software stack during all the required phases of the application: resource provisioning, configuration management, application deployment, software updates, monitoring, and access control. Among its features, it offers ease of use by having a wizard where the user can define the stack step by step and defining the Virtual machine resources assigned for each layer.

Configuration and initial customisation is handled through chef recipes like Ironfan does, this is quite useful as users can do testing of their recipes on their local machines through debugging tools like Vagrant [28] and later move their recipes to their stack in OpsWorks.

2.3.3 Cloudify:

Cloudify [29] is an effort of offering an OpenPaaS solution for companies who want to customize freely their services in the public cloud of their choice. The solution offered by Cloudify is very similar to how VMware or Hortonworks manage their Hadoop clusters, as we saw previously. In this case, we specify how we deploy our application and the necessary services by specifying the contents of a recipe file. This file will be executed by the Cloudify software and will configure the desired services on your public cloud. Also it has a monitoring application that lets you track the status of your applications in your cloud. This is possible due to the fact that during the configuration phase, Cloudify installs the Cloudify agents that send this data. The source code is publicly available



```
application {
        name="petclinic"
        service {
                name = "mongod"
        }
        service {
                name = "mongoConfig"
        }
        service {
                name = "apacheLB"
        }
        service {
                name = "mongos"
                dependsOn = ["mongoConfig", "mongod"]
        }
        service {
                name = "tomcat"
                dependsOn = ["mongos","apacheLB"]
        }
}
```

Figure 7: Cloudify overview

2.5 Closed PaaS vs Open PaaS

In PaaS platforms, we can make a distinction on two different views of defining PaaS; which are as follows:

Table 1: Comparison on types of PaaS

PaaS Type:	Description:
OpenPaaS	Supports configuration and upgrade of your own services on your VMs. Allows complete freedom for the user to define and specify their services in any cloud. Normally they are compatible with IaaS providers. There exist multiple techniques and technologies to attain this. Companies offer already their own solution. This would be the case of Cloudify, as we described previously.
ClosePaaS	Supports a fixed set of services. Not capable of customize your own services. Limited freedom as the provider only supports a fixed set of services under certain technologies. The users of these platforms need to follow the provider's specifications about how they support the applications to be deployed on their cloud infrastructure. This could be the case of Heroku [30] and Google app engine [31].

Chapter 3 - Problem Description:

In this chapter we will discuss the problem and motivation behind our efforts to realise KTHFS on a PaaS environment.

Launching and configuring nodes

KTHFS is composed by multiple services, like the underlying MySQL Cluster that stores the name nodes metadata; which interact among them in order to manage its underlying distributed file system. In this case, we need to launch nodes and configure them depending of their roles plus we must ensure that no conflicts arise when starting those services.

Normally, in Amazon EC2's case, a user would request a virtual machine with the virtual image they desire through the Amazon management console. It is believed that this process is not efficient due to how the management console is implemented. Submitting a query for the list of Virtual Images in Amazon take time to be processed, this is due to the amount of images available in the system. Later when the Virtual Machine is allocated, the client will access it through a secure tunnelling protocol like SSH and configures the software by manually running the necessary binaries and commands on the operating system.

Support multiple cloud IaaS providers

In order to reach a great number of potential clients, we need to make our distributed file system to be usable in different platforms. Most companies do not have resources to use public cloud infrastructures and possibly have their own private cloud infrastructure based in solutions like OpenStack, running at their own premises.

So in this case, it is important to offer some kind of solution that would allow users to have the flexibility of deciding whether to use private or public clouds infrastructures.

Support limitations of private cloud infrastructures

In the case of public cloud infrastructures, there is direct access to the nodes due to the fact that they are automatically allocated a public IP address which allows direct connectivity. On the other hand, private cloud infrastructures do not initially support this option as not all companies have a large pool of public IP addresses in order to allocate them to every virtual machine launched. It is important to consider this if we want our solution to be feasible in a real non ideal environment.

Automatisation of Clusters

Hadoop deployment can become tedious and complicated to accomplish when we are configuring a huge cluster of possibly hundredths of nodes. It requires a lot of time and resources to launch and manage a Hadoop cluster manually. In order for users to use our proposal, we need to facilitate the usage of this platform, to avoid misuse and to reduce management and deployment time.

In this case, we had to design a solution which will be attractive for the users who will use our platform. We need to think of a way that will allow users to configure and launch their own KTHFS clusters.

Chapter 4 - KTHFS Orchestration Architecture:

In this section, we will discuss the design and details behind the Orchestration tools for KTHFS. Our proposed architecture shall be capable of offering effectively Open PaaS capabilities.

4.1 Goals:

Our orchestration architecture aims to achieve the following objectives

I. Increase the ease of use of the KTHFS platform.

II. Automatic provisioning of a KTHFS cluster.

III. Freedom to configure a KTHFS cluster.

IV. Define a DSL that will allow users to define their own KTHFS cluster in order to interact with the system.

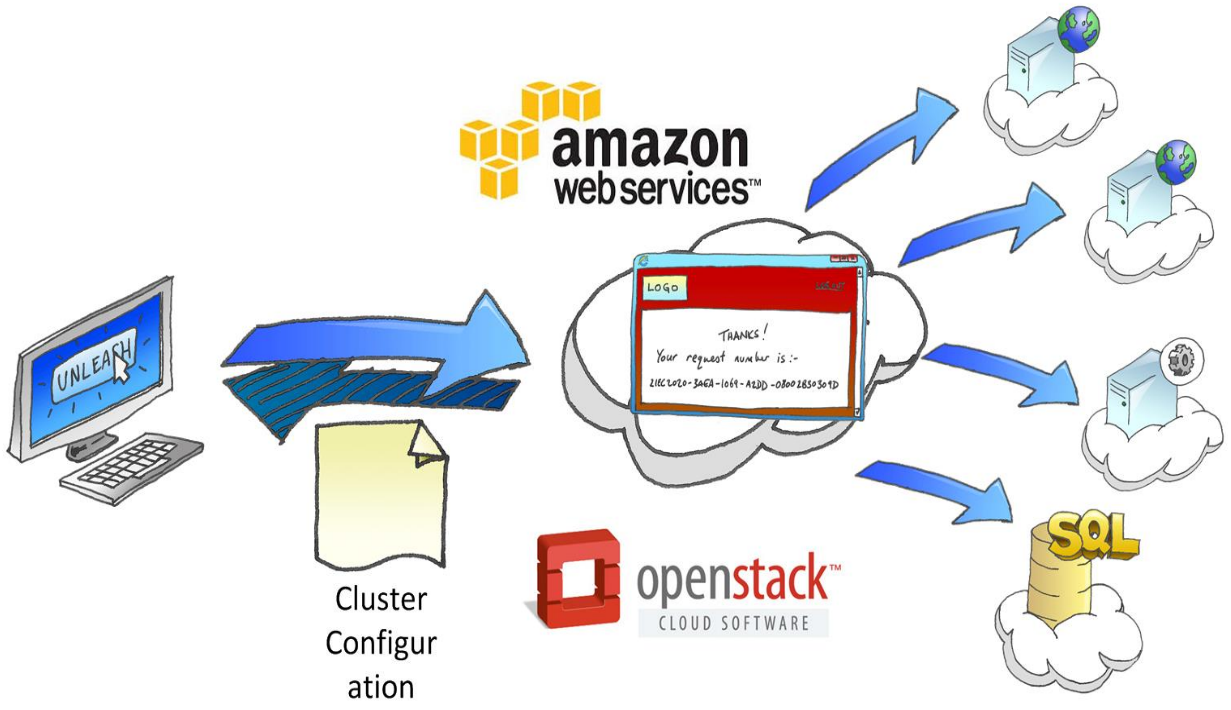


Figure 8: General Overview of orchestration

4.2 Analysis of Orchestration solutions:

In this section we will give a brief overview of what it is referred by the term orchestration, the process it usually follows and an overview of the actual orchestration languages that exist and are used for general purpose applications.

4.2.1 Orchestration:

Orchestration is a concept enabling the automatization, coordination and management of complex systems. With the interest of service provisioning in cloud environments, this term has become quite popular in the cloud community, mainly when it comes to define cluster configuration with the services to be run on the machines under cloud environments.

Therefore, an orchestration Domain Specific Language (DSL) allows the users to describe their clusters in a virtual media, normally in a specific file format; that it is interpreted by their orchestration platform and later on maps the necessary phases and actions from the information provided by the user in order to provision the services in the cluster.

As any language in real life, it needs to follow the rules of the syntax and semantics for that language; implying that we need to follow its specific syntax in order to communicate and interact with the properties of the system.

4.2.2 Structure of the process:

Understanding the contents of a file based on specific language syntax and later produce some results involves the following phases:

1. Parsing of the file.
2. Check if the syntax corresponds with the grammar of the language.
3. Map the contents of the file with functions to be done by the architecture and fetch the parameters needed by those specified by the user.
4. Run the required functions in order to fulfil the purpose behind that file.

4.2.3 Orchestration Languages in Cloud Computing:

There are a series of multiple alternatives to choose from when describing a cluster to be deployed in the cloud in Open PaaS. Here we list some of the approaches that someone can choose when designing their cluster.

Table 2: Orchestration technologies

DevOps Platform	Prog. Framework	Orchestration
Predefined AMIs	EC2/Jenkins/etc	Scripts
Chef	Full Ruby DSL	Ironfan DSL
Puppet	Limited Ruby DSL	Mcollective API
Bosh	YML	CloudFoundry
Pallet	Clojure DSL	Clojure DSL
Cloudify	Groovy DSL	Groovy DSL
JClouds	Bash	Java

If we consider the previous proposed alternatives on the market, generally they show a dependency on other technologies (normally programming languages or markup languages) in the underlying layers of the architecture. This is clear in the case of Ironfan which in its underlying software stack works on top of chef which, at the same time; uses Ruby to specify the contents of the files. Also, we can see that most of the proposed solutions by different companies follow a similar approach when automatizing the provisioning of their clusters and applications in IaaS providers.

The user initially needs to write a configuration file which will contain all the necessary information related to their cluster. This file will contain information related to the virtual machine requirements in their favourite cloud provider which all the nodes will deploy plus specific configuration parameters like the number of nodes, roles and security groups [32], which we can see from the following sample:

```
Ironfan.cluster 'big' do

  cloud(:ec2) do

    defaults

    availability_zones ['us-east-1d']

    flavor              'm1.xlarge'
    backing               'ebs'
    image_name            'natty'
    bootstrap_distro      'ubuntu10.04-ironfan'
    chef_client_script    'client.rb'
    mount_ephemerals(:tags => { :hadoop_data => true, hadoop_scratch => true })
  end
  environment             :prod
  facet :namenode do
    instances             1
    role                  :hadoop_namenode
    role                  :hadoop_secondarynn
    # the datanode is only here for convenience while bootstrapping.
    # if your cluster is large, set its run_state to 'stop' (or remove it)
    role                  :hadoop_datanode
  end


  facet :jobtracker do
    instances             1
    role                  :hadoop_jobtracker
  end


  facet :worker do
    instances             30
    role                  :hadoop_datanode
    role                :hadoop_tasktracker
  end
  cluster_role.override_attributes({
      # No cloudera package for natty or oneiric yet: use the maverick one
      :apt    => { :cloudera => { :force_distro => 'maverick',  }, },
      :hadoop                 => {
        :java_heap_size_max  => 1400,
        # NN
        :namenode             => { :java_heap_size_max => 1400, },
        :secondarynn          => { :java_heap_size_max => 1400, },
        # M
        :jobtracker           => { :java_heap_size_max => 3072, },
        :datanode             => { :java_heap_size_max => 1400, },
        :tasktracker          => { :java_heap_size_max => 1400, },
        # if you're going to play games with your HDFS, crank up the rebalance speed
        :balancer => { :max_bandwidth => (50 * 1024 * 1024) },
        # compress mid-flight (but not final output) data
        :compress_mapout_codec => 'org.apache.hadoop.io.compress.SnappyCodec',
        # larger s3 blocks = fewer tiny map tasks
        :s3_block_size        => (128 * 1024 * 1024),
      }
    })
```

The full example can be found in the appendix.

The configuration parameters usually appear following a pattern, if we follow the specific structure syntax that Ironfan uses from the previous example, we can extract the following abstractions:

- **Cloud:** This abstraction will allow the client to define basic parameters of the nodes in the cluster. In this case, it will contain the ID of the Virtual Machine Image that will be fetched when launching the nodes, the hardware resources for that node and the region/zone where the nodes will be deployed. In Amazon EC2's case, they handle different regions depending of the location of the data center.

- **Roles:** The roles indicate what part of the system or application the node is in charge of. Clients can define a series of basic roles that all the virtual machines will have by default and on top of this roles, clients can be more concrete by specifying more specific roles for each node launched in the cluster. In the previous sample, this would be the case of "hadoop_namenode" to refer the namenode role in HDFS or "hadoop_datanode" to refer the datanode role in HDFS.

- **Overriding Attributes:** In the Ironfan approach, there is an additional notion to override attributes when configuring the services on the nodes. This is inherited from the Chef framework which allows users to include extra configuration parameters when running the recipes. For example, a user could store a basic chef recipe that by default install Oracles Java Development Kit (JDK) but if the client prefers to use Open JDK, he can override this option by specifying the version of JDK they prefer through the override property. This is achieved by the metaprogramming properties in Ruby [33] that allow Ruby programs to modify their data during runtime.

Once this file is loaded in the system, it is interpreted by the orchestration architecture and maps the necessary tasks with the specified attributes in the cluster file during the provisioning process of the cluster.

4.3 Designing our solution:

After defining our problem statement and defined some general guidelines on what type of architecture we should work towards at, we will discuss the different design decisions we made when selecting the different technologies for our PaaS solution.

From the initial constraints of our problem, we decided to implement our own solution using the following set of technologies:

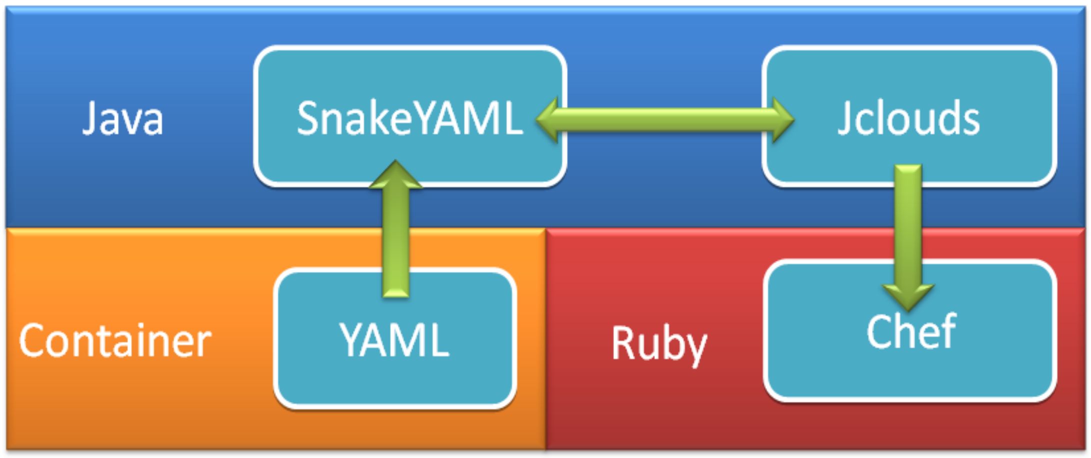


Figure 9: KTHFS orchestration stack

On the following sections we will explain the decision carried when selecting this set of technologies.

4.3.1 Chef:

Chef [26] is a technology that works on top of the Ruby programming language which, thanks to its metaprogramming properties [33], makes it a great tool for provisioning. This property allows users to specify default configuration parameters and later allow them to extend their functionality by overriding its parameters. Chef works with two different types of approaches; you can run the chef locally through chef-solo or you can fetch the recipes and cookbooks remotely using a chef server.

Chef has some key components that we mostly use in our proposed solution:

- **Recipes:** a recipe manages which resources we wish to install in the system (Software packages, configure accounts, etc) and how we manage them in order to configure the desired elements for the environment. The recipes are specified in a run list which runs the recipes in the order that they are defined.

- **Cookbooks:** refers to a collection of recipes which are focus towards to a role or configuration.

It was decided to work using the Chef approach for the following reasons:

- Well documented and good support from the company and the community.
- Very mature technology.
- It is possible to debug the recipes through Vagrant [28], a software solution that allows creation and configuration of reproducible virtualized environments. It allows the users to simulate the type of machine they want to work with and specify the software you want to install. In this case, we can use it to debug chef recipes by having a starting vagrant with an environment that contains the KTHFS software and the chef recipes for our system.

4.3.2 Jclouds:

Jclouds [34] is an open source API where anyone can contribute to add more features to their extensive API. It is a library which allows users to manage, configure and launch virtual machines in a large variety of cloud providers like Amazon EC2, OpenStack, Rackspace or even Microsoft Azure. Recently with version 1.6, it also allows the possibility to work with Chef and configure nodes with Chef Servers. The library is written and supported in Java but also has the option to use it with Clojure [35].

Their API works with an abstraction referred to as compute service which acts as a wrapper over all the different cloud providers APIs supported in their library [36]. This way, only by using one abstraction we can easily launch and configure nodes at multiple cloud providers. In addition, if users need to work with more complex scenarios like the ones we handled in this thesis, there is the option to use more concrete abstractions and functions for specific providers like OpenStack.

Quite recently, this project has been proposed to form part of the apache foundation community and now it is in an incubator state. We decided to go with the Jclouds library in Java for the following reasons:

- The monitoring application of KTHFS is a Java Web application; it would be easier to enhance this web application with Jclouds functionality.
- Good community behind its development.
- Tested at multiple providers, mainly Amazon EC2 and OpenStack which were the main platforms we were focusing. We also considered using Ironfan, their solution worked perfectly on Amazon EC2 but still did not have full support for OpenStack based cloud platforms which we were also interested to give support. For that reason, we discarded the Ironfan option and decided to use Jclouds.
- The support for multiple cloud platforms also allows us to extend our solution to be deployed in other cloud providers in the future.
- Large base of code examples, which allowed us to quickly learn the main elements of the platform, plus they come with a well-documented API.
- With 1.6 they added better support to chef functionality allowing people to work with Chef-solo or with their Chef servers.

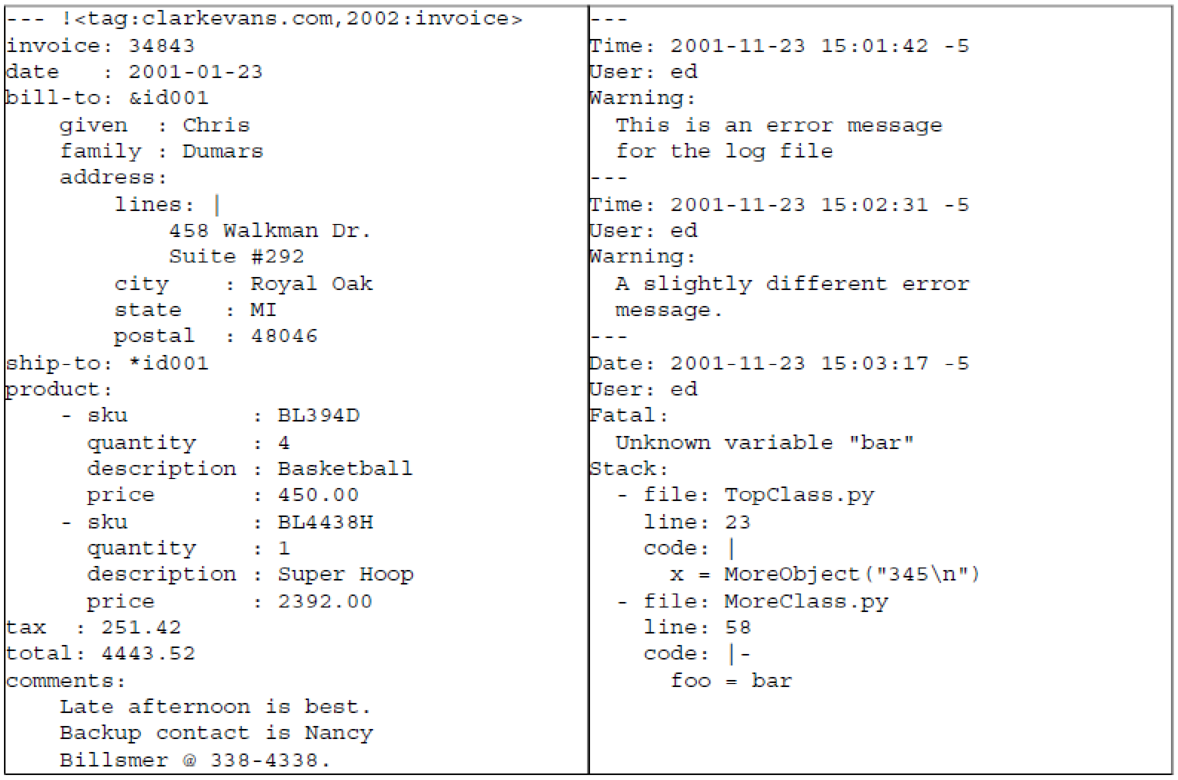
- It allowed us to handle node information after deploying a group of nodes which is useful as it gave us their private and public IP addresses. This way we can keep information of the nodes IP for interconnecting later the Hadoop nodes.

4.3.3 YAML:

Now that we had defined the core technologies when coming to provisioning nodes, we had to specify a container where the user will define the parameters and configurations of the cluster. Our main focus was to find a file specification powerful enough to be extended if needed plus at the same time allowed us to keep things as simple as possible. There exist many alternatives when it comes to data formats. In this case, we have mark-up technologies like XML [37] which are widely used and alternatives like JSON when it comes to this.

On the other hand, you can also find rare alternatives which are probably less known like the case of YAML [38]. YAML (YAML Ain't Markup Language) is a nice alternative to the other well extended formats like the ones previously mentioned.

YAML was first proposed by Clark Evans in 2001. The first draft proposed the guidelines for a format which looked to define a data serialization language which was human readable and at the same time computational friendly. Its core concepts were inspired of programming languages like C, Perl and Python plus notions of other file formats like XML and data format of the email. A sample file would look like the following, taken from the YAML specifications:



```
--- !<tag:clarkevans.com,2002:invoice>
invoice: 34843
date   : 2001-01-23
bill-to: &id001
    given  : Chris
    family : Dumars
    address:
        lines: |
            458 Walkman Dr.
            Suite #292
        city    : Royal Oak
        state   : MI
        postal  : 48046
ship-to: *id001
product:
    - sku         : BL394D
      quantity    : 4
      description : Basketball
      price       : 450.00
    - sku         : BL4438H
      quantity    : 1
      description : Super Hoop
      price       : 2392.00
tax  : 251.42
total: 4443.52
comments:
    Late afternoon is best.
    Backup contact is Nancy
    Billsmer @ 338-4338.
```

```
---
Time: 2001-11-23 15:01:42 -5
User: ed
Warning:
  This is an error message
  for the log file
---
Time: 2001-11-23 15:02:31 -5
User: ed
Warning:
  A slightly different error
  message.
---
Date: 2001-11-23 15:03:17 -5
User: ed
Fatal:
  Unknown variable "bar"
Stack:
  - file: TopClass.py
    line: 23
    code: |
      x = MoreObject("345\n")
  - file: MoreClass.py
    line: 58
    code: |-
      foo = bar
```

Figure 10: YAML sample

Here we can appreciate the simplicity of this format compared to other options like XML which tend to be quite complicated and tedious. By reading the file we can get a clear idea of the type of representation we are handling.

To summarize, it was decided to go for YAML as the container for our orchestration platform mainly because:

- It is very simple to define your own data structure.
- The syntax is simple and powerful to use
- Less tedious compared to other technologies like XML which tend to take a lot of time to be formalised and be defined correctly.
- Less error prone compared to JSON when handling large definitions of a data structure where it is easy to miss a comma or parenthesis. This is usually translated in long hours of debugging and finding the missing element in the definition.
- Designed to be easily mapped with data types from high level languages like lists and arrays.

4.3.4 SnakeYAML:

Once we decided to use YAML as the format where user will define their clusters for KTHFS, we needed a way to obtain the information defined in the cluster so we could map it with the necessary functions in our system. In this case, we needed to find a way so we could parse the contents of the cluster, although YAML is not so widely extended; we managed to find implementations of YAML parsers in Java. In the end, it was decided to use SnakeYAML [39], a Java library for parsing YAML; due to:

- It is still maintained by the community, this was a key decision as it showed that the library was still updated.
- Extensive and detailed documentation of its API, this helped us to get started and know how to use this technology quite quickly.
- It had support for object serialization which helped us to map quicker the contents of the cluster into memory.

- It offers a simple syntax mapping when checking the contents of the file by specifying the class library we are using to map the contents of the file with a Plain Old Java Object (POJO).
- It supported the specification of YAML 1.1.

4.4 Defining our own DSL:

In this section we will explain the process through we followed to specify the DSL for our KTHFS platform based on the software stack we defined in the previous section.

4.4.1 Defining the Cluster Structure:

Defining the grammar of a DSL resembles something similar when defining the functions and functionalities of an API [40]. We need to be very careful on how we define it as if we want people to use it should be:

- Easy to understand.
- With the conventions of the syntax, the user can realise what he can do with the functions.
- Anything that we add to the syntax should mean that it is a functionality necessary and needed a 100%. If that is not the case, we should not include it in the version and leave it for the future. This avoids problems of breaking previous code or using functionalities that could be remove in a future version or not even work as expected.

So our initial thinking process should be how do we define a cluster? What information should it contain? Should it be extensible? What are the key elements of a cluster? Based on our previous analysis on similar solutions, like the case of Ironfan; we identified the following elements for our abstraction:

- **Cluster:** A cluster is a group of nodes defining the system we are launching in the cloud and the software to be run on them.
- **Provider:** A provider will define the necessary information we need to launch a node in the cloud. It will specify the cloud infrastructure where we are launching the cluster, the region, the virtual image to use and the instance type.
- **Services:** In a cluster we identify that multiple services will be shared in the cluster. For example in our KTHFS implementation we work with different services which interact between them, in this case the MySQL Cluster and KTHFS nodes.
- **Roles:** For each service, we can divide it in several sub roles which will be in charge of a certain functionality offered by the service they are mapped to. For example, in the case

of defining a MySQL Cluster we have different components: Management Server (MGM), Database nodes (NDB) and MySQL Daemon (MYSQLD).

Also other aspects were considered when it came to add future support in extended functionality and abstractions in cloud providers:

- **Security Groups:** It allows the definition of firewall rules that all the nodes will share if they form part of the same group.
- **Specific Ports:** Allow authorization of specific ports that might be requested by the user of our platform and are not collected in the services associated to KTHFS.
- **Global Services:** Services that might be shared across all the nodes. In this case, for example we could support this type of services:
    - chefClient
    - chefServer

With all this information we proceed to define a draft structure of our orchestration platform, which in YAML will look like this:

```
## YAML Template.
---
!!se.kth.kthfsdashboard.virtualization.clusterparser.Cluster
  name: test
  environment: dev
  globalServices:
    [ssh, chefClient]
  authorizePorts:
    [ssh, chefClient, chefServer, http&https, webserver]
  authorizeSpecificPorts:
    [3306, 4343, 3321]
  provider:
```

```
  name: aws-ec2
  instanceType: m1.medium
  image: eu-west-1/ami-ffcdce8b
  region: eu-west-1
  zones:
    [eu-west-1a]
##lists of groups, with the roles the nodes will have and open ports
nodes:
- securityGroup: ndb
  number: 2
  roles:
   [MySQLCluster-ndb]
  authorizePorts:
   [MySQLCluster-ndb]
- securityGroup: mgm
  number: 1
  roles:
   [MySQLCluster-mgm]
  authorizePorts:
   [MySQLCluster-mgm]
- securityGroup: mysql
  number: 1
  roles:
   [MySQLCluster-mysqld]
  authorizePorts:
   [MySQLCluster-mysqld]
- securityGroup: namenodes
  number: 2
  roles:
   [KTHFS-namenode]
  authorizePorts:
   [KTHFS-namenode]
- securityGroup: datanodes
  number: 2
  roles:
   [KTHFS-datanode]
  authorizePorts:
   [KTHFS-datanode]
##Override chef attributes for roles:
chefAttributes:
##Include JSON data:
```

Figure 11: KTHFS cluster definition sample

Which in SnakeYAML, we will map the information through the following class relationship diagram by the SnakeYAML parser libraries.

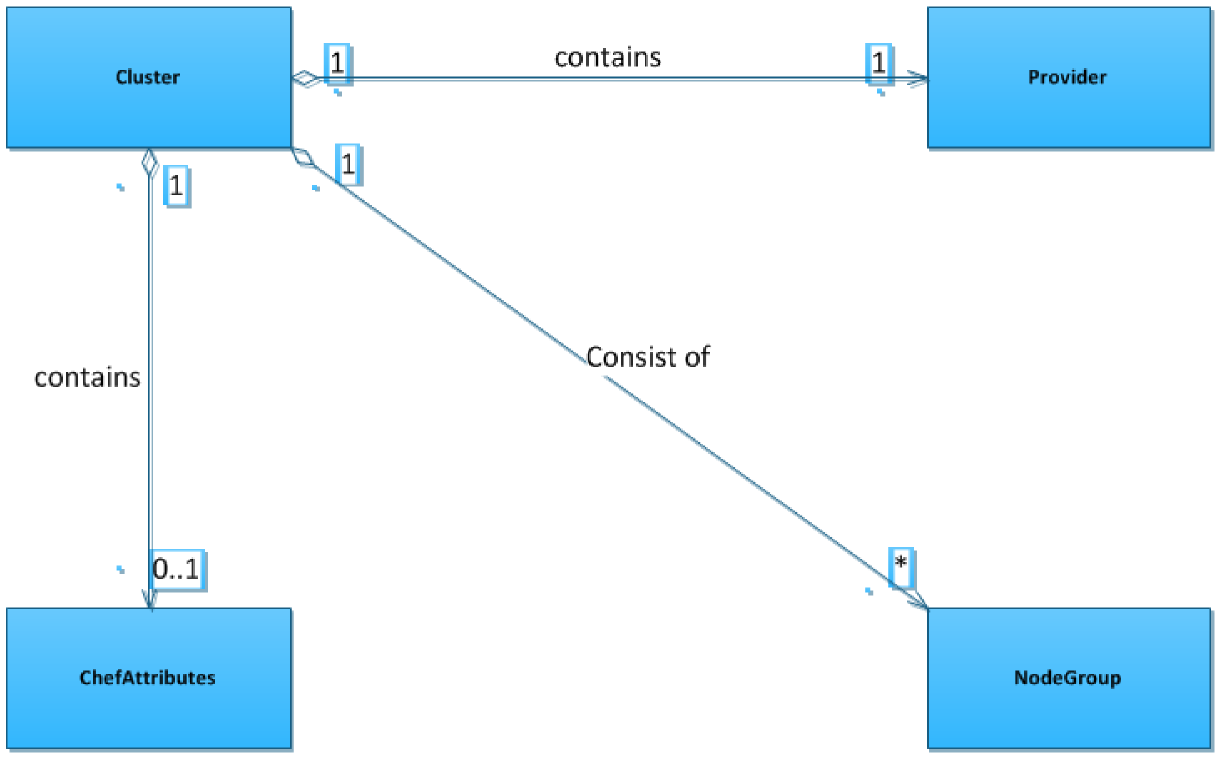


Figure 12: Cluster relationship class diagram

4.4.2 Defining the grammar syntax:

The KTHFS Orchestration language should have support for simple definitions a user will need in the context of a cluster definition, plus know the necessary elements KTHFS needs. Further in this section, we will formalise our DSL.

In this case, considering the previous class diagram when a user speaks of a cluster it will refer to:

- Name to identify the cluster
- The type of environment of how the cluster is being handled.
- Simple services of the KTHFS context that a user might need.
- Open the ports for the desired services of the user.
- Specify ports that the user might want to open.

Also it will handle the following abstractions when it comes to compose a Cluster:

Provider:

When speaking of a provider, the user will be referring to the following terms:

- Name of the provider where we are going to launch the cluster.
- Type of Instance, refers to the virtual machine hardware configuration of the provider.
- Virtual image which will be referred using the ID from the cloud provider we are using.
- Region indicates the data center where we will launch the nodes. This will be referred to by the user in relation to the cloud infrastructure where they are launching their nodes.
- Zones indicate which specific physical area of the data center we are launching the nodes, this is used by Amazon EC2 to specify a specific area in their region.

Nodes:

When referring to nodes, the user will be referring to a set of nodes that will specify the configuration of services for a specific role:

- Name used to define the security group in the cloud provider.

- Number of nodes composing the group.
- KTHFS roles handled by the nodes in the group.
- Ports to be open for the group based on the roles

Chef Attributes:

In the future, a user would like to override the attributes for the chef recipes. In this case, it would be simple enough to provide their own JSON with the desired attributes to be overridden for the role and append it when generating the scripts for the nodes.

4.4.3 ANTLR Formalisation:

ANTLR (ANother Tool for Language Recognition) [41] allows us to define the grammar for our own language. In this case, it allows us to formalize the syntax of the elements that compose the building blocks for our DSL.

As our language works on top of YAML, we need to also respect the syntax of the YAML specification which in this case the SnakeYAML parser already verifies when parsing YAML files.

Our syntax for KTHFS DSL will look as follows:

```
grammar Cluster;


cluster
        : DOCUMENT_START
        EOF PACKAGE
        EOF name
        EOF environment
        EOF globalServices
        EOF authorizePorts
        EOF authorizeSpecificPorts
        EOF provider
        EOF nodes
        EOF chefAttributes
        EOF DOCUMENT_END
;


name
        : 'name:' WHITESPACE IDENTIFIER_CLUSTER_NAME;
```

```
environment

          : 'environment:' WHITESPACE IDENTIFIER_CLUSTER_ENVIRONMENT;


globalServices

          : 'globalServices:' EOF WHITESPACE openarray_definition
SUPPORTED_GLOBALSERVICES_LIST closearray_definition

          | /* empty */

;


authorizePorts

          : 'authorizePorts:' EOF WHITESPACE openarray_definition
SUPPORTED_GLOBALSERVICES_LIST closearray_definition

          | /* empty */

;


authorizeSpecificPorts

          : 'authorizeSpecificPorts:' EOF WHITESPACE openarray_definition
PORT_NUMBER_LIST closearray_definition

          | /* empty */

;


provider

          : 'provider:' EOF WHITESPACE provider_name

          EOF WHITESPACE provider_instanceType

          EOF WHITESPACE provider_image

          EOF WHITESPACE provider_region

          EOF WHITESPACE provider_zones

;
```

```
nodes
	: 'nodes:' EOF (DASH WHITESPACE securityGroup)+;


securityGroup
	: 'securityGroup:' WHITESPACE [a-zA-Z0-9_.-]* EOF
	'number:' WHITESPACE WHITESPACE [0-9]+ EOF
	'roles:' EOF WHITESPACE WHITESPACE openarray_definition KTHFS_ROLE_LIST
closearray_definition EOF
	'authorizePorts:' EOF WHITESPACE WHITESPACE openarray_definition
KTHFS_ROLE_LIST closearray_definition EOF
;


chefAttributes
	: 'chefAttributes:' EOF (DASH WHITESPACE attributes)*;


attributes
	: 'role:' kthfsRole EOF
	WHITESPACE WHITESPACE 'chefJson:' /* JSON grammar*/
;


KTHFS_ROLE_LIST
	: kthfsRole kthfsRoles;
;


kthfsRoles
	: ',' kthfsRole kthfsRoles
	| /* empty*/
;
```

```
kthfsRole
          : 'KTHFS-namenode'
          | 'KTHFS-datanode'
          | 'MySQLCluster-mgm'
          | 'MySQLCluster-ndb'
          | 'MySQLCluster-mysqld'
;

provider_name
          : 'name:' WHITESPACE ('aws-ec2'    | 'openstack-nova');

provider_instanceType
          : 'instanceType:' WHITESPACE
          ('c1.medium'
          | 'c1.xlarge'
          | 'cc1.4xlarge'
          | 'cc2.8xlarg
          | 'cg1.4xlarge'
          | 'hi1.4xlarge'
          | 'hs1.8xlarge'
          | 'm1.large'
          | 'm1.medium'
          | 'm1.small'
          | 'm1.xlarge'
          | 'm2.2xlarge'
          | 'm2.4xlarge'
          | 'm2.xlarge'
```

```
        | 'm3.2xlarge'
        | 'm3.xlarge'
        | 't1.micro'
        | [0-9]*)
;

provider_image
        : 'image:' WHITESPACE (region '/' [a-zA-Z0-9_.-]*
        | [a-zA-Z0-9_.-]*)
;

provider_region
        : 'region:' WHITESPACE region
;

region
        : 'ap-northeast-1'
        | 'ap-southeast-1'
        | 'ap-southeast-2'
        | 'eu-west-1'
        | 'sa-east-1'
        | 'us-east-1'
        | 'us-standard'
        | 'us-west-1'
        | 'us-west-2'
        | [a-zA-Z0-9_.-]*
;
```

```
provider_zones
        : region [a-z];

SUPPORTED_GLOBALSERVICES_LIST
        : service services
        | /* empty */
;

services
        : ',' service services
        | /* empty */
;

/*list of supported services */
service
        : 'webServer'
        | 'chefClient'
        | 'chefServer'
        | 'http&https'
        | 'ssh'
;

/*Range of Ports */
PORT_NUMBER_LIST
        : port ports
        | /* empty */
;
```

```
ports
        : ',' port ports
        | /* empty */
;

port
        : ([0-9]{1,4}|[1-5][0-9]{4}|6[0-4][0-9]{3}|65[0-4][0-9]{2}|655[0-2][0-9]|6553[0-5])
;

openarray_definition
        :'[';

closearray_definition
        :']';

IDENTIFIER_CLUSTER_NAME
        : [a-zA-Z0-9_.-]*;

IDENTIFIER_CLUSTER_ENVIRONMENT
        : 'dev'
        | 'production'
;

DOCUMENT_START
        : '---';

PACKAGE
        : '!!se.kth.kthfsdashboard.virtualization.clusterparser.Cluster'
```

```
;


DOCUMENT_END
          : '...';
EOF
          :'\n';


DASH
          :'-';
WHITESPACE
          : ' ' | '\t';
```

Figure 13: DSL syntax definition for KTHFS

Chapter 5 - Prototype Implementation:

In order to evaluate the virtualization functionality using our DSL for automatizing cluster provisioning, we needed to provide proof of concept based on two simple scenarios of interaction with our platform.

5.1 Overview of the Architecture

The design of the architecture for how a user will use the KTHFS can be specified in the following diagram:

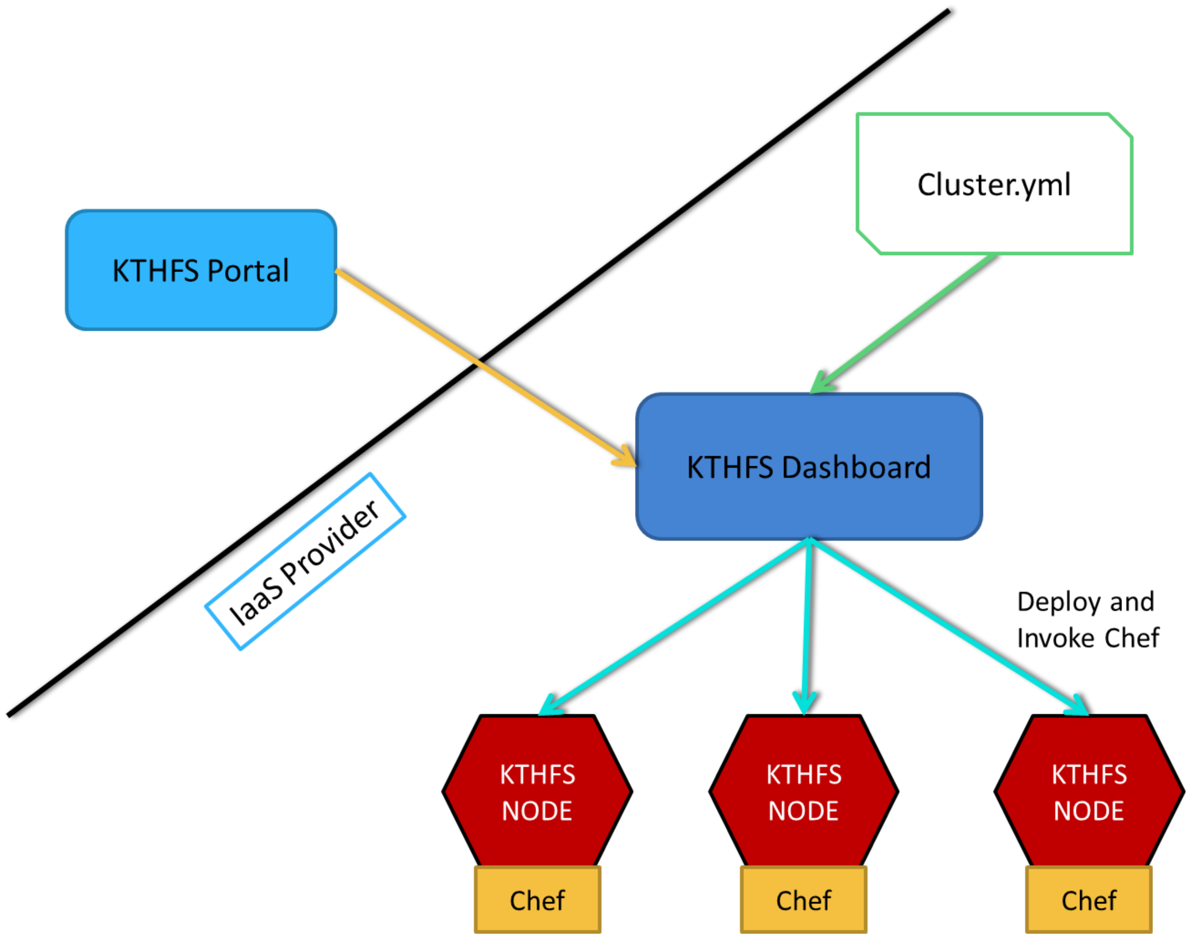


Figure 14: Orchestration architecture overview

The process of how the user will interact will be composed of the following phases:

1. The user uses an external web application, KTHFS Portal. This application will use the user's credentials to connect to Amazon EC2 or OpenStack to launch and configure the KTHFS Dashboard in a virtual machine. After finishing the configuration, it will give back to the user a link to connect to the newly created Dashboard.

2. With the newly created Dashboard, the user authenticates in the application and configures the credentials in order to communicate with Amazon EC2 or OpenStack. Then, it will upload a cluster definition file and the system will proceed with the automatic provisioning of the KTHFS cluster, defined in the uploaded file. Once it finishes, the information of the newly created cluster will appear in the KTFHS Dashboard.

5.2 Proof of Concept: KTHFS Portal

Our first artefact involves a simple use case where a user launches our platform. To have an easy interaction with KTHFS, we designed a simple web application where the user will input its user credentials of the cloud provider they are willing to use and we can actually support. This would launch the KTHFS Dashboard web application in a virtual machine in their desired cloud infrastructure so later own they could manage, monitor and configure the nodes required by KTHFS.

In addition to this simple scenario, it helped us to get used on developing a Java Server Faces (JSF) web application using Primefaces [42][43] and start learning to launch a simple configuration using Jclouds in a cloud provider like Amazon EC2 or OpenStack. Primefaces is a JSF component framework that offers a large variety of components which allow a richer user experience when interacting with web applications. So the main goals with this proof of concept were:

5.2.1 System goals:

- **Ease of use:** Offer a simple application where a user could have a simple starting point in order to start using KTHFS.

- **User friendly:** It should make the user comfortable when using our application, so in this case the design of the user interface should be attractive.

- **Functional:** It should deploy the dashboard correctly.

5.2.2 Platform Architecture:

Our first step before starting to implement the web application was to identify for our scenario the use case related to it. In this case based on the previous description, we design the following use case diagram:

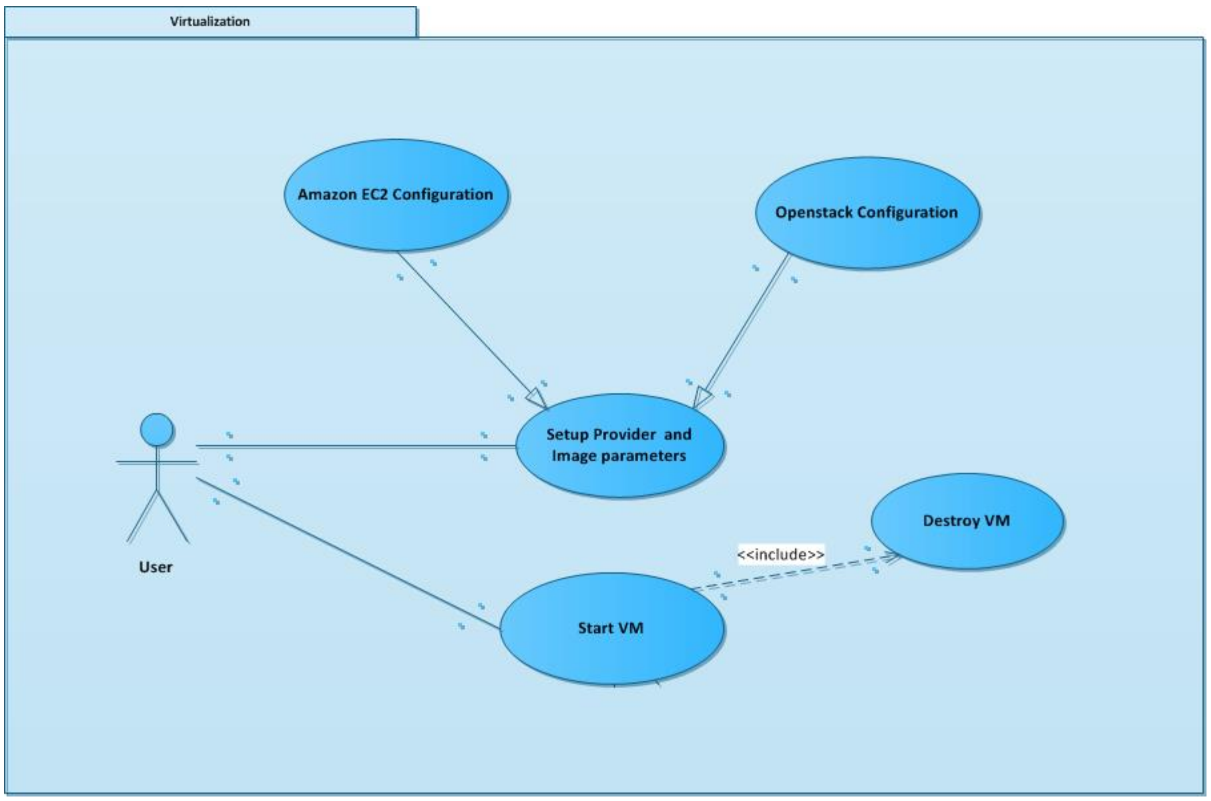


Figure 15: KTHFS Portal use case diagram

From the use case diagram, we can abstract the following information and identify the following functionality our web application will need to handle:

- An authenticated user will fill the information needed to login into the cloud infrastructure, in this case between Amazon EC2 or OpenStack
- In the case of OpenStack, it will also include the login credentials for the virtual image used in a private cloud.
- Start the virtual machine and provisioning of the KTHFS Dashboard.
- If desired, have the option of destroying the virtual machine.

A detailed description of the use cases can be found in the appendix.

Designing the Web application:

After defining the use case scenarios for our KTHFS Portal, we modelled the interactions of the user with our platform and identified the necessary relations so we can model the web application using the Model View Controller architectural pattern (MVC) [44]. It is a quite useful pattern for software development as it allows the separation of the components associated with the presentation, business logic and modelling involved:

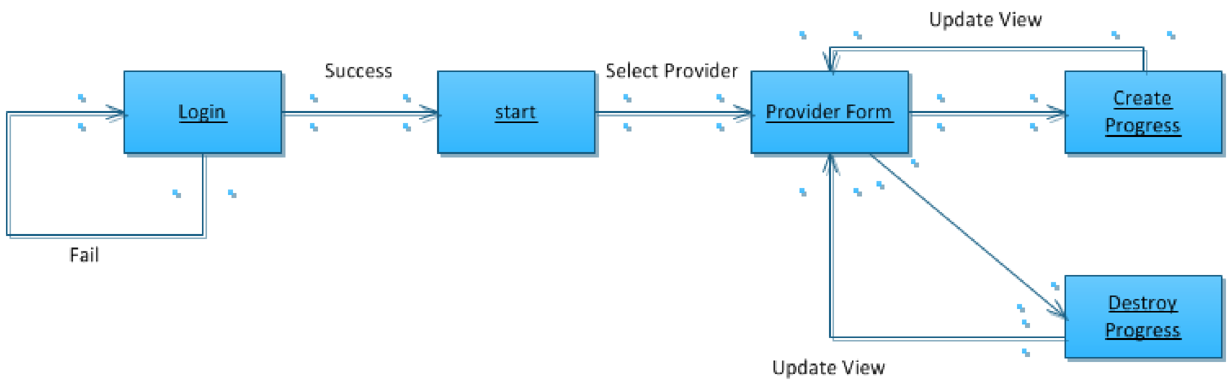


Figure 16: KTHFS Portal page flow

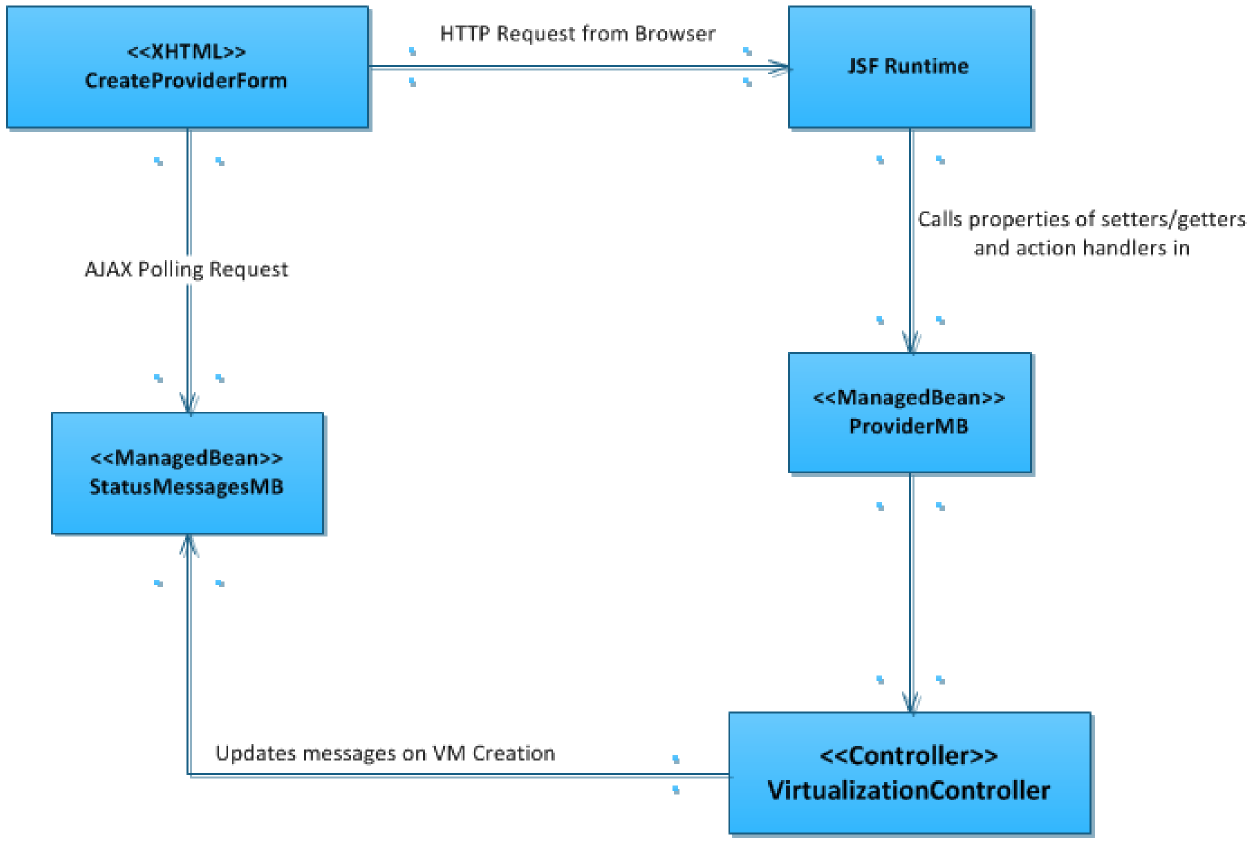

HTTP Request from Browser
<<XHTML>> CreateProviderForm
JSF Runtime
Calls properties of setters/getters and action handlers in
AJAX Polling Request
<<ManagedBean>> ProviderMB
<<ManagedBean>> StatusMessagesMB
Updates messages on VM Creation
<<Controller>> VirtualizationController

Figure 17: Instance Virtualization flow

Configuration phases:

When we launch the virtual machine with the dashboard web application, we configure the software in multiple phases. We decided to follow this approach using several phases mainly due to limitations we had with the OpenStack platform.

If we wanted to launch a node in OpenStack, the Nova module will automatically provide a virtual machine with a private IP address. Later, when Jclouds tried to SSH to the virtual machine it failed as it did not have direct connectivity to the Virtual Machine.

To solve this, we needed an intermediate procedure that will allocate a public IP from a pool of IP addresses available for the OpenStack region after successfully deploying the node. Of course, in order for this to work, the user of our platform must configure their region to have at least one public IP address in their pool; otherwise we cannot add the new IP

After, we will configure the necessary scripts that we run via SSH using chef on the virtual machine by submitting the script and running it.

So the process will be as follows:

- Launch node, create the security group and open the ports.
- With the node running, if it is OpenStack; we allocate the public IP address
- We create the script that will configure and download the necessary software in the node and install chef. The software will be downloaded from a remote Git hub repository from SICS, in this case the *“kthfs-dash”* recipes
- We submit another script that will run chef with the run list of required recipes.

Handling Heavy Processes:

Launching and configuring nodes take time; we need to wait for the virtual machines to be deployed and running by the Cloud provider. In order to keep track of the progress of the dashboard setup, we added status messages to be shown in the web application during the dashboard deployment. That way the user could keep track of the deployment status.

5.3 Proof of Concept: KTHFS Virtualization Support

Our second scenario was to extend the main functionality of KTHFS Dashboard. It is a monitoring JSF web application which is in charge of monitoring the status of the involved nodes in the KTHFS system.

In our case, the main objective and purpose of this thesis was to implement PaaS functionality in this platform. For this, we will make use of the technologies we explained in the previous sections and show the use of our KTHFS DSL to deploy a functional cluster in Amazon EC2 or Openstack.

5.3.1 Goals:

- **Functional:** Enhance the KTHFS Dashboard functionality with virtualization support; this involves automatic deployment and configuration of a KTHFS cluster.
- **High Performance:** The system should be capable to deploy a cluster in a reasonable amount of time.
- **Fault Tolerant:** The system should be capable of handling failure situations when they occur in the deployment.

5.3.2 Implementation:

The virtualization functionality for KTHFS Dashboard depends of other functionalities which we need to design and implement. The parsing of YAML files is already managed by the SnakeYaml library and the function calls to the cloud provider is managed by the Jclouds library.

This means that we need to work on how we can combine together these components and create the necessary interactions to achieve the provisioning of a KTHFS cluster in the cloud. In the previous scenario with the KTHFS Portal, it was easy to handle as were interested in provisioning only one node in the cloud. But in this scenario we need to work with a more complex configuration as we need to launch and provision multiple nodes.

Mapping the Cluster file:

With SnakeYAML, it is easy to load POJOs (Plain Old Java Objects) in memory as it supports Java object serialization/deserialization from YAML files. Simply, we need to define the Java classes that represent the blocks used by the file in the form of basic Java representations or Java references to objects.

Still this does not solve how we translate the information which is linked with essential elements of our KTHFS DSL grammar. We need to map the grammar elements to its data type's reference in our system, mainly associate the ports to open with the roles defined for the KTHFS roles and map the necessary chef recipes to be executed by the each group of nodes.

In order to do this, we designed a dictionary abstraction that will map the KTHFS roles and common services with the ports needed to be open and be able to choose between the TCP or UDP ports of the roles. A Hash map data structure will be used to store the desired roles depending of the protocol and in order to make it thread safe it will be an immutable object.

On the other hand, we manage the mapping of the roles with their respective recipes in another abstraction that we will explain in the following sections when generating the chef scripts.

Virtualization management:

Provisioning a cluster of nodes is a similar process to the one we described in the scenario of KTHFS Portal. The main difference is that we are handling multiple nodes at once and it is possible that we may encounter incompatibilities if we do not install the software in the right order.

In this case, apart from installing the necessary libraries and launching the nodes for the KTHFS platform; we need to configure the nodes following the order imposed by the dependencies between the roles in the services of KTHFS.

The provisioning process can be divided in the following steps:

- Create the necessary security groups defined in the cluster file.
- For each of those groups, create the nodes in the cloud.
- We create the script that will configure and download the necessary software in the node and install chef. The software will be downloaded from a remote Git hub repository from SICS, in this case the *"kthfs-pantry"* project.
- Once all the nodes are created, configure the nodes in the correct and no conflicting order.

The orders in which services need to be deployed are as follows:

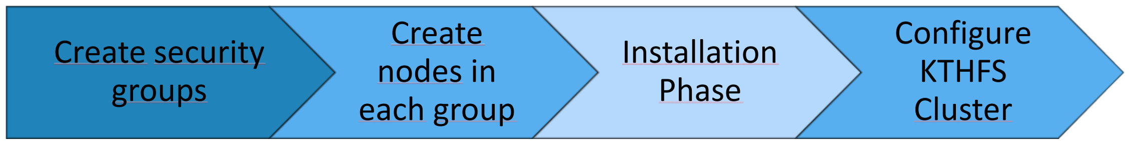


Figure 18: KTHFS orchestration process

Where in the last phase we need to install the roles in the following order:

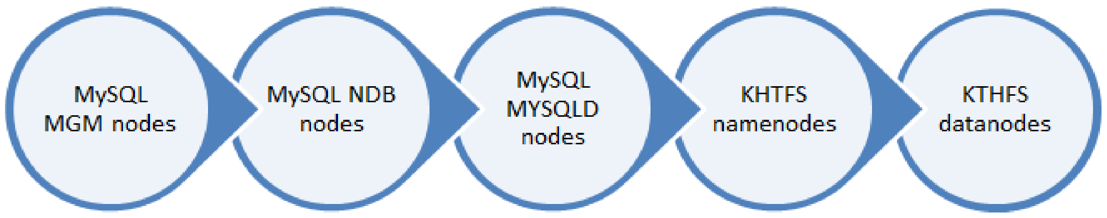


Figure 19: KTHFS node configuration phases

The whole virtualization process is defined in the ClusterProvision class where we defined all the necessary methods to create security groups and nodes, launch the nodes and prepare the configuration recipes for the node. In addition it sends information to KTHFS Dashboard about the progress of the cluster deployment.

KTHFS ScriptBuilder:

In general, the scripts need to be customised for the specific node roles and for each specific node in the system. They all share a common structure in relation to the Chef JSON that needs to be executed when launching the Chef-solo command. On the other side, they all require different configuration parameters and specific parameters of the node like the IP of the node where we are submitting the script. So it is important to design a component that allows us to generate the scripts to be executed in the nodes of KTHFS and also flexible enough to achieve this.

In this case, we designed a script generator using the Builder pattern [45] which allows us to obtain different types of script using different parameters for the KTHFS nodes. For now, three different types of script are generated:

- **Bootstrap:** It initializes the virtual machine with a preliminary setup by installing some basic dependencies and setting the git credentials in order to clone the KTHFS chef recipes from a remote git repository.
- **Install:** This script is in charge of downloading the necessary binaries for MySQL cluster and KTHFS into the virtual machine.
- **KTHFS script:** In this case, we specify inside the builder the skeleton of the script which a KTHFS node will run with the specific Chef recipes. Before submitting the script in the node, we will configure this script based on the role defined for that node.

Chapter 6 - Synchronous vs. Asynchronous Deployment:

Our first working version of the provisioning architecture was developed using the synchronous abstractions for node configuration from Jclouds API. With this abstraction, it allowed us to confirm that our architecture managed to provision a KTHFS cluster but we were limited with this approach, as our initial test results showed us.

If we follow our provisioning process in detail, we can identify that set of operations can be handled in parallel, sequentially or a mixture of both.

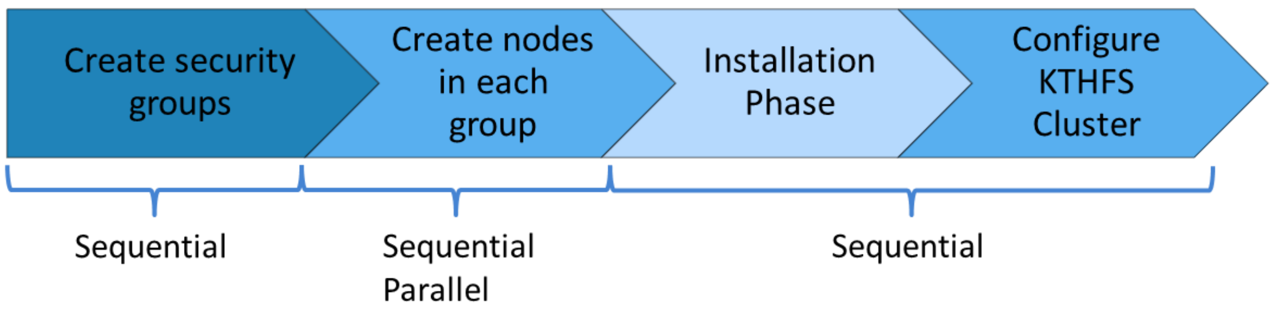


Figure 20: Operation maps during orchestration process

By how Jclouds handle node creations, we created all the nodes involved in that security group in parallel but we had to do it for all the security groups defined in our cluster, this meant that we created group of nodes sequentially. In the case of creating security groups, the rules submitted to handle the ports for the firewalls had to be generated one after the other.

Lastly, we had to provision the nodes sequentially. This was mainly because Jclouds provisioning methods were synchronous which their usage blocked the system until the operation was finished.

We saw in this case that the bottleneck in the performance of our system were the synchronous calls that limited our system. The culprits in this case were the last phases were we configured and created the scripts for the nodes. This process, as we mentioned previously; required for us to create a specific script for each node.

With this situation, a new problem arose and in this case we encounter **scalability issues**. We had now to design our solution so it would be capable of scaling in proportion to the nodes being launched. Analysing the Jclouds API, we discovered that the actual implementation of Jclouds 1.6.0 offered asynchronous node configurations. This appeared to be our best tool to solve our scalability issues and will give us a larger freedom when handling node configurations in the system, if we do a brief comparison:

Table 3: Synchronous and asynchronous comparison

Synchronous	Asynchronous
Advantages: ● Easier to handle	**Advantages:** ● Non-blocking. ● Usage of fewer resources compared to synchronous threads.
Disadvantages: ● Request blocks ● Slower to start ● It uses more resources when keeping track of the process**.**	**Disadvantages:** ● Difficult to track when has the process finishes or something goes wrong. ● call-backs

Moving to asynchronous node provisioning appeared to be an appealing option as it enabled us to parallelise the number of nodes been configured in each phase and solve our scalability issues. Still new challenges appeared when using this approach, we now had to handle time synchronization problems of multiple asynchronous processes which might never finish in the desired time frame.

Therefore adapting our provisioning architecture to handle asynchronous node configuration became a new challenge which introduced new concepts and design decisions in our working components. We will discuss further in the following sections.

6.1 Thinking Asynchronously:

One of the most referred abstractions when handling asynchronous processing in distributed environments is referred as the ***Futures*** notion.

A **Future** abstraction refers to a computational result that is not available in the system mainly because the process has not been completed or its transfer through the network has not yet been completed. Still it is promised that in the end the result will be available or not [46]. Technically, we could refer a future as a process computation that will retrieve a result through a callback to the original thread.

The Java programing language offers abstractions and methods to work using **Futures** (java.util.concurrent.Future). Handling this programming abstractions seemed quite complex to understand directly as we had to work with asynchronous processes.

Jclouds asynchronous processing in this case differs from Java's offering by using a better and more refined abstraction offered by Google's Guava Libraries [47]. Working with Google's ListenableFutures abstractions appeared to be more powerful as it allowed us to attach runnable tasks that will run once we received a notification that the call finished plus extra transformations when working with multiple asynchronous calls like **chaining of futures.** This allowed for example, having multiple futures chained together allowed us to do several processing operations through this chain. Still we had to work with Google's approach mainly because Jclouds uses Google's Guava abstraction when handling asynchronous node configurations.

Unfortunately we will not go deeper on describing the power of Google's library, but numerous tutorials and blogs describe their utility and ease of building our own ListenableFuture abstractions. If the reader is further interested to hear from this abstractions, you can refer to the following useful links [48] [49].

6.2 Scaling Asynchronously:

Thanks to Jclouds Asynchronous abstractions, it was easy and fast to reuse most of our solutions designed for a synchronous configuration. Our main problem in this scenario was that in order to handle parallelization of node configurations in each phase we needed to consider the following issues:

- What happens if we have slow node configurations?
- How we ensure that we can move to the next configuration phase?
- How do we handle possible timeouts or nodes that never get configured?

Therefore in an asynchronous scenario and in our system, we need to carefully take care of these problems and so our system needs to consider the following additional properties:

- **Scalable**
- **Fault tolerant**

6.2.1 HDFS MapReduce:

We introduce a brief description of how the HDFS MapReduce framework works in order to give a better understanding of our asynchronous solution in our provisioning platform as it is a similar approach. We will not go deeply in Map Reduce itself as we want only to use its architecture to describe our architectural solution when handling asynchronous processes when provisioning nodes in the cloud, as this architecture is capable of offering scalability and fault tolerant properties.

Map Reduce [50] is a programming model for processing large data sets making use of parallel, distributed algorithms. It is composed of two phases: **Map** and **Reduce** inspired in the notion of Map/Reduce descriptions in functional programming.

The **Map** phase refers to the basic idea of filtering out all the unnecessary information or parameters stored in a HDFS cluster for our computation. The **Reduce** phase refers to the refinement of these preliminary results from the map phase in order to do further processing.

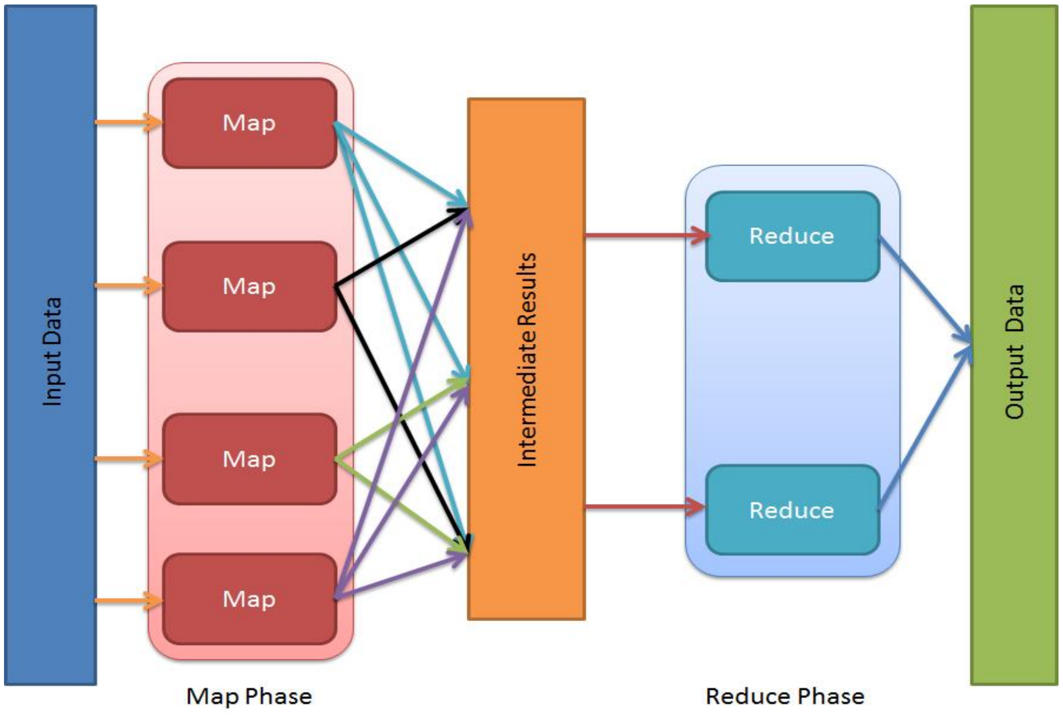


Figure 21: MapReduce workflow

6.2.2 Our Solution:

The Map Reduce workflow previously defined is similar to how we have designed our approach towards our asynchronous provisioning and it is useful to understand the process behind our implementation.

In our case we can define that each node configuration in essence acts like a Map Reduce workflow but more simplified as in our case, we will not be handling processing of big data sets and our reduce phase will not do further processing. In addition, we will reuse the Reduce phase to help us synchronize and track the process of the configuration phase.

In our case, a Node configuration phase workflow can be broken down to the following pattern:

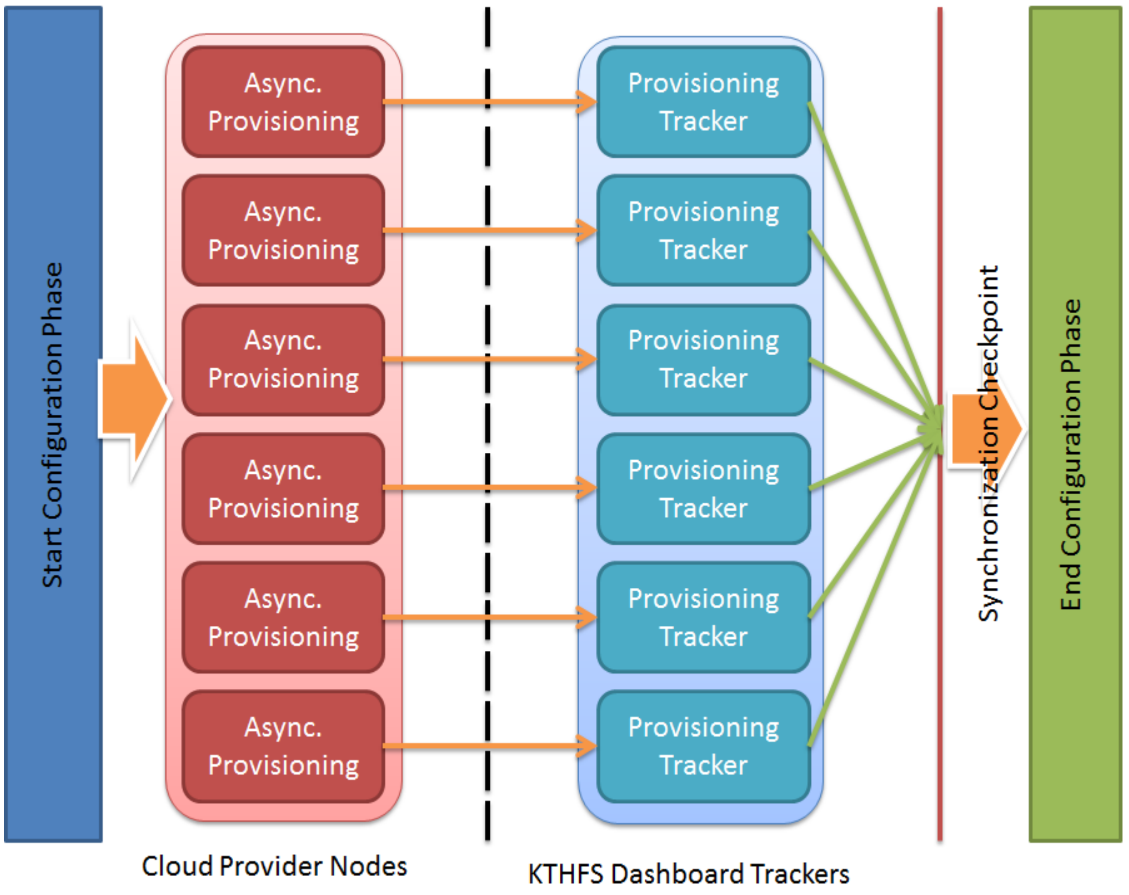


Figure 22: Asynchronous provisioning workflow

With the ListenableFutures abstraction model from Google's Guava Library, it is quite simple to attach a separate thread to work upon the results we gather after a node has finished its provisioning in the cloud. Still, we need to effectively ensure that all nodes reach a certain point before finishing this configuration phase and proceed to the next one.

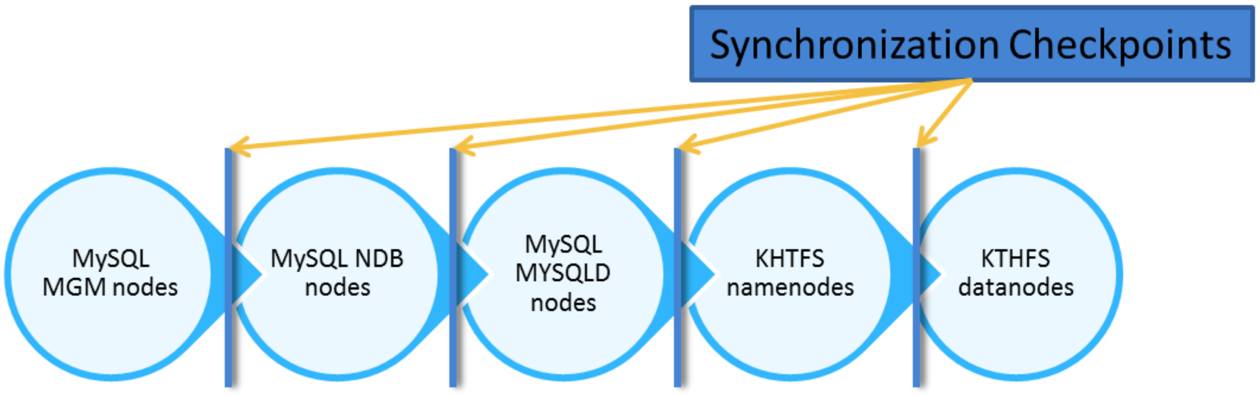


Figure 23: Synchronization checkpoints during provisioning

Synchronising multiple threads is always critical and special care is needed as with concurrent programming, it is possible to suffer unexpected behaviour in our systems when handling shared resources. To ensure that all threads meet the expected processing checkpoint, there exist multiple abstractions in java.util.concurrent package to handle multiple thread synchronisations:

- **CyclicBarrier:** Reusable abstraction to ensure that the computations do not proceed until an expected number of threads meet the barrier point [51].
- **CountDownLatch:** Similar to a Cyclic Barrier but in this case you cannot reuse the object once you create it [52].

Due to the nature of our problem, a Cyclic Barrier would have been ideal due to its reusability, as we are following the same workflow in multiple phases. The problem with this abstraction is that it is suitable if you expect the same number of threads to process the same work, which it is not our case. In each node configuration phase, we launch a variable number of nodes which differ in each phase in number so this abstraction is of no use for us. In that case we proceed to make use of a CountDownLatch to synchronise the threads.

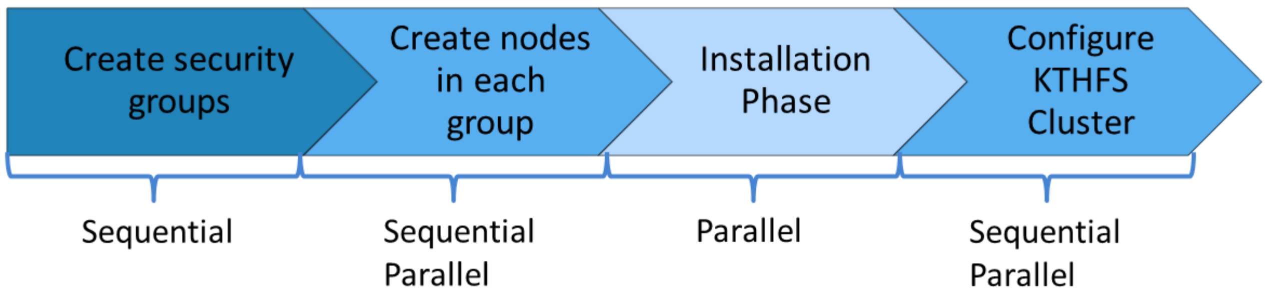


Figure 24: Operation maps during asynchronous provisioning

With this update, now we managed to make all the nodes in the system to download the necessary software in parallel, improving the provisioning of the nodes in our system. Also it allowed us to parallelize in a similar way to how Jclouds create nodes in the security groups, in the configuration of the KTHFS nodes. We could node configure nodes with similar roles in parallel.

Still we had to face with another problem, the possibility that nodes might not finish in time or not ending. One way would be to include a timeout and check which nodes completed their provisioning and evaluate if we need to launch again the provisioning in those slow nodes.

Using CountDownLatch's timeout we can achieve this and check which nodes need to be provisioned again. With this simple approach, we managed to obtain a limited fault tolerant solution to detect slow nodes.

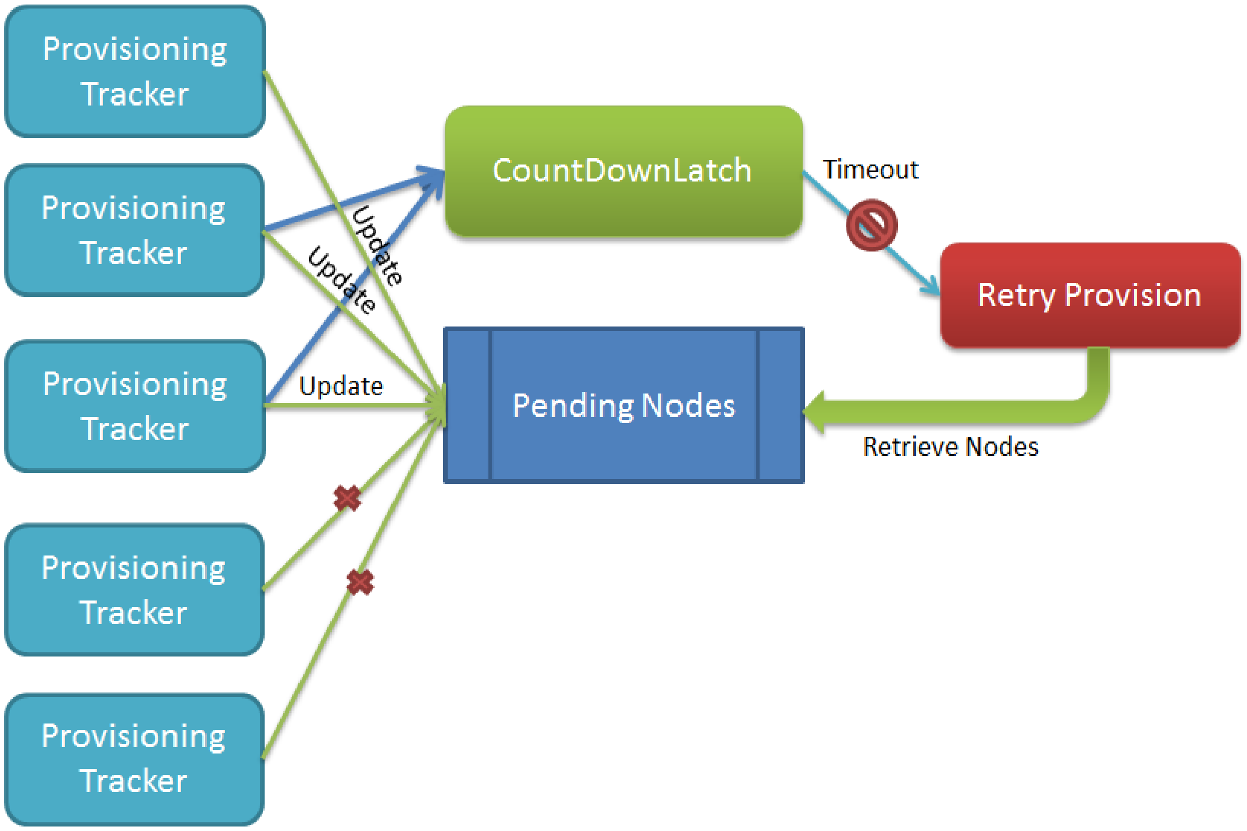


Figure 25: Fault Tolerant design for slow node provisioning

Chapter 7 - Evaluation:

In this section, we provide an evaluation of our orchestration solution by testing a real cluster structure of KTHFS and discuss its scalability and performance when it deployed a cluster in the cloud.

7.1 Test Scenario:

To test our PaaS provisioning in KTHFS we defined cluster file which will consist of 8 nodes. In this case the cloud infrastructure where we will deploy our cluster will be Amazon EC2. We will use a modified version of KTHFS Dashboard for our tests which contains the new functionality already implemented, forming part of the KTHFS Dashboard source code. We describe the following configuration (a detailed cluster file can be found in the appendix):

Table 4: Amazon EC2 instance configuration

Amazon EC2 Instance Size:	m1.large
Resources allocated for the node:	Processor Arch: 64-bit vCPU (Virtual CPU): 2 ECU (EC2 Compute Unit)*: 4 RAM: 7.5 GiB Instance Storage: 2 x 420 GB
Amazon AMI	Ubuntu 12.04 Desktop edition 64 bits.
Amazon Region	eu-west-1

**one ECU provides equivalent CPU capacity of a 1.0-1.2 GHz 2007 Xeon Processor [53].*

The cluster structure is defined as follows:

Table 5: Test cluster structure

MySQL Cluster:	
Security Group: ndb	Number of Nodes: 2 Roles: MySQL NDB

Security Group: mgm	Number of Nodes: 1 Roles: MySQL MGM
Security Group: mysqld	Number of Nodes: 1 Roles: MySQLD
KTHFS	
Security Group: namenodes	Number of Nodes: 2 Roles: namenode
Security Group: datanodes	Number of Nodes: 2 Roles: datanode

7.2 Synchronous Provisioning Deployment:

The first working implementation that we tested when we launched our test cluster in Amazon EC2 handled the configuration phase of the nodes by using the Jclouds synchronous access when handling node configurations after launch. This allowed us to individually configure the scripts specifically for each role before submitting them in the nodes.

In this case our evaluation criterion to test our provisioning solution was to measure the amount of time required to launch our cluster defined previously for our testing scenario. A test run was validated if we managed to retrieve the nodes information in the cluster section on the KTHFS Dashboard. With this first version, a run of eight nodes took approximately **1 hour 20 minutes in the best case.**

These results allowed us to confirm that our orchestration solution achieves our initial goal, but still; this is not satisfactory in a production environment. Also we were limited in our deployment scheme as the configuration phase of the nodes was done **sequentially** which reduced our throughput on the number of nodes we launched.

In real cluster configurations, users are expected to launch larger clusters with a greater number of nodes (we can reach clusters of possibly hundreds of nodes). Still provisioning is an operation that takes time of course but our solution is still open for further optimizations.

7.3 Asynchronous Provisioning Deployment:

The first version of our provisioning system, limited our throughput because we were limited to synchronous nodes deployment in a sequential manner. As we explained previously, this was the main bottleneck in our previous solution.

On the other hand, we had these limitations due to the fact that we needed to personalise each script for each node in the system and if we used the synchronous abstractions, due of their blocking nature; no more tasks could be launched from that point.

In order for our system to be feasible on production environments, we needed to increase the throughput of the number of nodes we could configure in parallel. Still our parallelization was limited due of the fact that we had to handle node configuration in phases. If we were able to configure nodes all the nodes in the specific phase in parallel, we would gain a considerable boost in throughput compared to the initial sequential approach.

In this case, we tested our asynchronous solution similar to the synchronous approach; we deployed the same cluster configuration as in the previous scenario and waited until we retrieved information of the nodes in the cluster in the dashboard. With this alternative, we managed to launch a cluster of eight nodes in **approximately 32 minutes and 22 seconds.**

We continued doing further testing and we launched a bigger cluster using the same configuration as previously but duplicating the number of nodes in each group, so in this case with a 16 node cluster it took **around 51 min.**

7.4 Performance Benchmark:

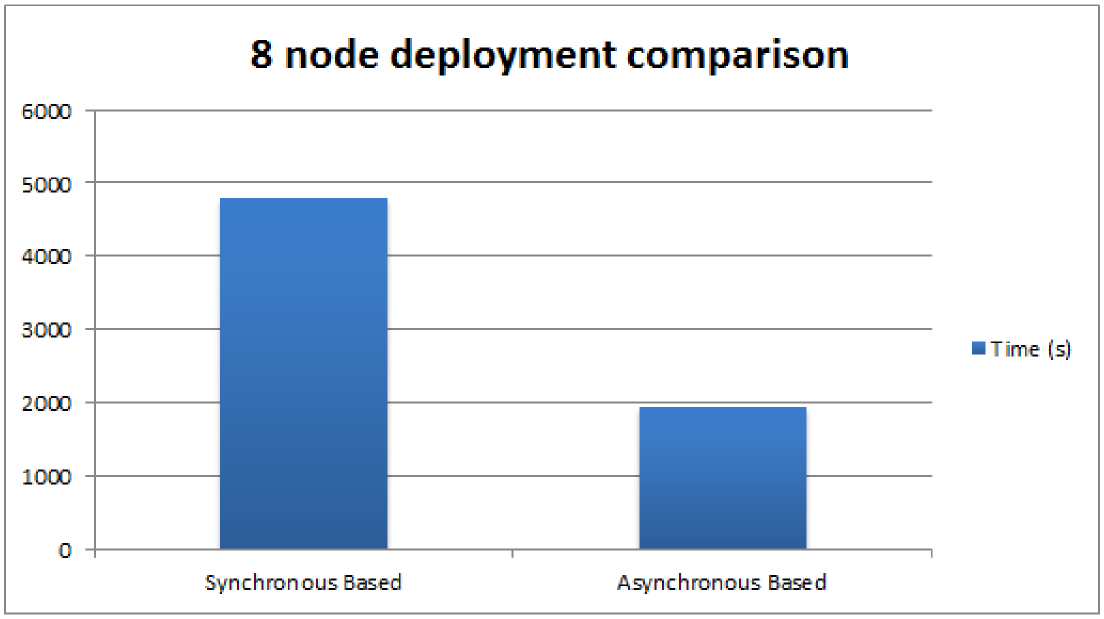


Figure 26: Comparison sync. vs. async. 8 node cluster

The first graph shows visually the performance when comparing our lazy approach with the asynchronous version. We can assure that we have obtained a performance gain of **59.5%,** making a major reduction on the time needed to deploy a simple cluster using our PaaS support.

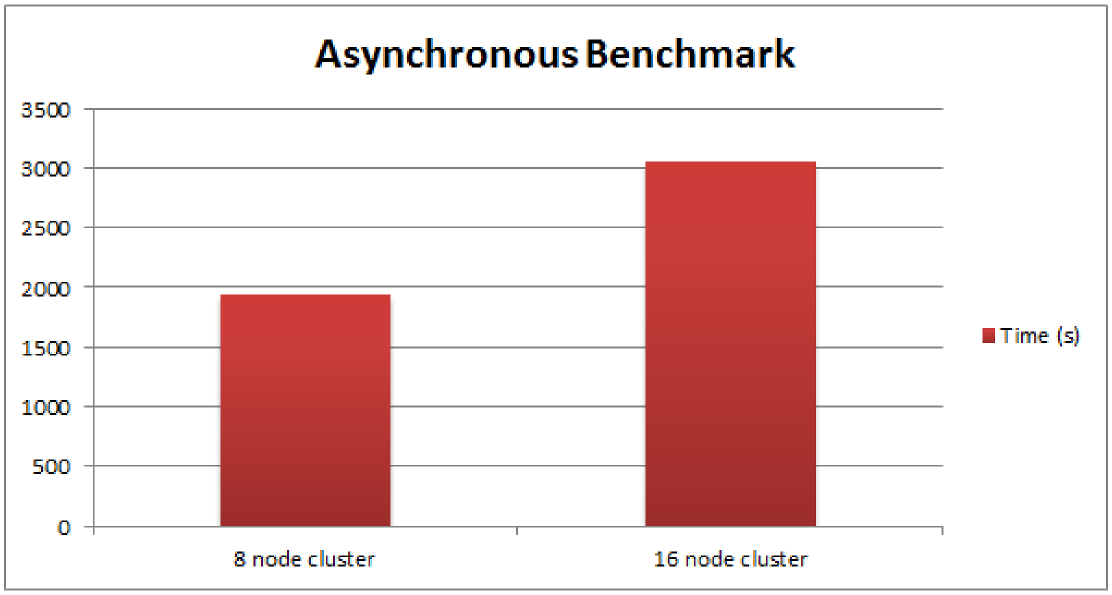


Figure 27: Comparison async. provision 8 vs. 16 nodes

On the second graph, we compared the deployment times of launching eight nodes and sixteen nodes. We saw that when we doubled in this case the number of nodes, the time required to deploy the whole cluster increased by **36.5%**

7.5 Optimizing even further:

Although with our asynchronous solution we managed to improve our provisioning performance and more importantly enabled to scale the configuration phase in proportion to the number of nodes. This was not enough compared to the first solution and still we could reduce the time our cluster was deployed. Several insights were considered that would considerably help attain this reduction:

- **Predefined Virtual Images:** Creating predefined Virtual Images with the software KTHFS uses, would drastically reduce the provisioning time. This is mainly because during the installation phase, we need to download the necessary binary which in the case of MySQLCluster it takes most time to download as it is one of the biggest dependencies we need to obtain. In addition, this approach could allow us to discard the installation phase which will lead us directly to the configuration phases of KTHFS.

- **Parallelization of the create node phase:** One inconvenience of Jclouds when it comes to deploy nodes in the cloud, is that this process is done synchronously and only launches multiple nodes in parallel for the number of nodes involved in a security group.

Due to the blocking nature of synchronous calls, we have a similar problem with our synchronous provisioning solution for KTHFS. If we could wrap this process using Listenable Futures in a similar approach as our asynchronous deployment, probably we could launch multiple node groups in parallel.

Due to time constraints, these optimizations were not tested in our prototype as initially our goal is to provide an orchestration deployment model for KTHFS and test the automatization of the provisioning process using this model. These optimizations can be introduced later in the architecture of KTHFS.

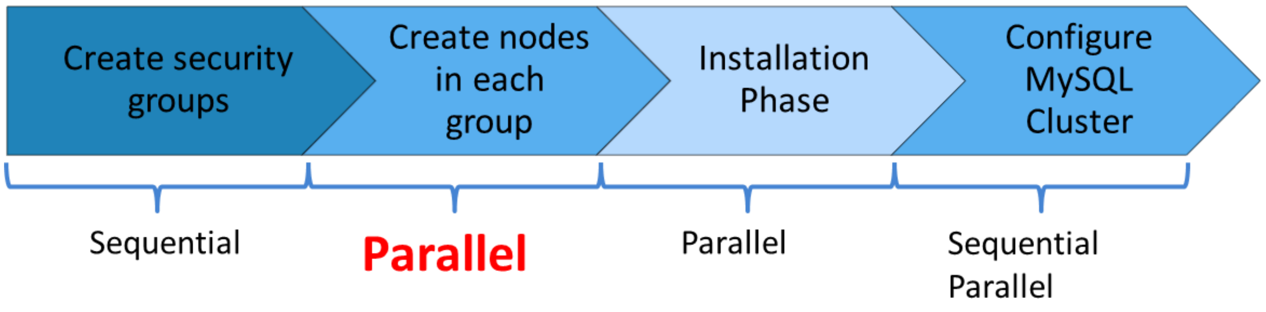


Figure 28: Operations map during provisioning with optimizations

Chapter 8 - Conclusions and Future Work:

In this section we will finalize with some final insights about our work and we will discuss further on the future work from a starting point of this thesis.

8.1 Conclusions:

In this report, we discussed the details and implementation of an Orchestration platform in order to deploy a KTHFS cluster in a series of selected cloud providers. As mentioned in the architecture section, the goal of this work was to define an automatic provisioning platform so users could deploy and use KTHFS in both public clouds like Amazon EC2 and private clouds like OpenStack. As we saw with our proposed solution and the evaluation results we managed to automatically provision our desired KTHFS cluster. Although, initially the task at hand was well defined; after getting our first implementation working, we faced new problems that needed to be take care of. Still having these problems that we did not expect initially, we managed to define the solutions and even implement the most critical features in short period of time.

Our orchestration platform is capable of provisioning a KTHFS cluster using a simple syntax and under a reasonable amount time for our testing purposes. Furthermore, we see that with our asynchronous architecture; we manage to improve the deployment time compared to our initial version and still we managed to identify future optimizations that will probably improve the deployment time of KTHFS.

8.2 Future Work:

Following we will identify future improvements for our orchestration solution and are intended to be completed in later stages of the project.

8.2.1 Implement further optimizations

We previously identified a series of optimizations for our solution based on our initial results with our provisioning system. This involved in modifying how we handled the creation node phase and having predefined virtual images with all the necessary software stored in it already.

Definitely, implementing these in the KTHFS Dashboard will improve further more the throughput performance of the nodes we can provision and reduce the deployment time for a cluster of a large volume of nodes.

8.2.2 Elasticity

In PaaS platforms like Google app engine [31] allow elasticity of the running applications in their cloud infrastructure. This means that when there is a peek of demand of a certain application, the underlying system will allocate more resources for that application in order to allow more users accessing it.

The next step for our proposed solution would be to design a mechanism that will allow a KTHFS cluster to dynamically allocate nodes depending of the volume of information stored in the distributed file system. This will offer a more scalable and dynamic file system.

8.2.3 Detect bad provisioning of virtual machines

In our fault tolerant design, we specified a solution that allowed the KTHFS Dashboard to provision slow nodes. Of course this solution does not detect nodes that fail to configure correctly. The next step to improve the performance of our fault tolerant approach is to identify mechanisms that will allow us to detect erroneous nodes.

8.2.4 Automatic reconfiguration of crashed nodes

It is possible that under extreme situations multiple nodes may crash unexpectedly. In those cases, the system should be able to handle this scenario and keep running without any problems. With the monitoring solution that the KTHFS Dashboard has, we could implement mechanism that when they detect that nodes start to fail unexpectedly, it will start to provision the failed nodes to take place of the previous crashed node.

Appendix A: Detailed Use Cases for KTHFS Portal

Detailed Information of the use cases:

Name	Setup Provider and Image Parameters
Actors	Initiated by Authenticated User
Flow of events	1. The user wants to use KTHFS 2. See enters the system 3. Fills the cloud provider for Amazon EC2 or OpenStack. 4. Fills the Credentials for the Virtual Image in the case of using OpenStack.
Entry Conditions	User successfully logs in the Web Portal in the create VM form.
Exit Conditions	The user fills the necessary information to launch and configure the KTHFS Dashboard in one of the Cloud Providers.

Name	Start VM
Actors	Initiated by user
Flow of Events	Once the user has filled all the necessary configuration parameters it presses the launch button.
Entry Conditions	The user has filled the start VM form.
Exit Conditions	The user has pressed the launch button and the configuration phase starts.

Included Cases:

Name	Destroy VM
Actors	Initiated by user
Flow of events	User selects and fills the destroy VM form
Entry Conditions	User is in the destroy VM form
Exit Conditions	The user has pressed the destroy button and the destroy phase starts.

Appendix B: Test Cluster File

```
## YAML Template.
---
!!se.kth.kthfsdashboard.virtualization.clusterparser.Cluster
  name: test
  environment: dev
  globalServices:
    [ssh, chefClient]
  authorizePorts:
    [ssh, chefClient, chefServer, http&https, webserver]
  authorizeSpecificPorts:
    [3306, 4343, 3321]
  provider:
    name: aws-ec2
    instanceType: m1.large
    image: eu-west-1/ami-ffcdce8b
    region: eu-west-1
    zones:
      [eu-west-1a]
  ##lists of groups, with the roles the nodes will have and open ports
  nodes:
  - securityGroup: ndb
    number: 2
    roles:
     [MySQLCluster-ndb]
    authorizePorts:
     [MySQLCluster-ndb]
```

```
 - securityGroup: mgm
   number: 1
   roles:
    [MySQLCluster-mgm]
   authorizePorts:
    [MySQLCluster-mgm]
 - securityGroup: mysql
   number: 1
   roles:
    [MySQLCluster-mysqld]
   authorizePorts:
    [MySQLCluster-mysqld]
 - securityGroup: namenodes
   number: 2
   roles:
    [KTHFS-namenode]
   authorizePorts:
    [KTHFS-namenode]
 - securityGroup: datanodes
   number: 2
   roles:
    [KTHFS-datanode]
   authorizePorts:
    [KTHFS-datanode]
 ##Override chef attributes for roles:
 chefAttributes:


...
```

Appendix C: Ironfan Cluster File

```
Ironfan.cluster 'big' do
  cloud(:ec2) do
    defaults
    availability_zones ['us-east-1d']
    flavor              'm1.xlarge'
    backing             'ebs'
    image_name          'natty'
    bootstrap_distro    'ubuntu10.04-ironfan'
    chef_client_script  'client.rb'
    mount_ephemerals(:tags => { :hadoop_data => true, hadoop_scratch => true })
  end

  environment           :prod

  role                  :systemwide
  role                  :chef_client
  role                  :ssh
  role                  :nfs_client
  role                  :volumes
  role                  :minidash

  role                  :hadoop
  role                  :hadoop_s3_keys
  role                  :tuning
  role                  :jruby
  role                  :pig
  recipe                'hadoop_cluster::config_files', :last

  facet :namenode do
    instances           1
    role                :hadoop_namenode
    role                :hadoop_secondarynn
    # the datanode is only here for convenience while bootstrapping.
    # if your cluster is large, set its run_state to 'stop' (or remove it)
    role                :hadoop_datanode
  end

  facet :jobtracker do
  instances            1
  role                 :hadoop_jobtracker
end

facet :worker do
  instances            30
  role                 :hadoop_datanode
  role                 :hadoop_tasktracker
end

cluster_role.override_attributes({
    # No cloudera package for natty or oneiric yet: use the maverick one
    :apt    => { :cloudera => { :force_distro => 'maverick',  }, },
    :hadoop                => {
      :java_heap_size_max  => 1400,
      # NN
      :namenode            => { :java_heap_size_max => 1400, },
      :secondarynn         => { :java_heap_size_max => 1400, },
      # M
      :jobtracker          => { :java_heap_size_max => 3072, },
      :datanode            => { :java_heap_size_max => 1400, },
      :tasktracker         => { :java_heap_size_max => 1400, },
      # if you're going to play games with your HDFS, crank up the rebalance speed
      :balancer => { :max_bandwidth => (50 * 1024 * 1024) },
      # compress mid-flight (but not final output) data
      :compress_mapout_codec => 'org.apache.hadoop.io.compress.SnappyCodec',
      # larger s3 blocks = fewer tiny map tasks
      :s3_block_size       => (128 * 1024 * 1024),
    }
  })

# Launch the cluster with all of the below set to 'stop'.
#
# After initial bootstrap,
# * set the run_state to :start in the lines below
# * run `knife cluster sync bonobo-master` to push those values up to chef
# * run `knife cluster kick bonobo-master` to re-converge
#
# Once you see 'nodes=1' on jobtracker (host:50030) & namenode (host:50070)
# control panels, you're good to launch the rest of the cluster.
```

```
  #
  facet(:master).facet_role.override_attributes({
      :hadoop => {
        :namenode    => { :run_state => :start, },
        :secondarynn => { :run_state => :start, },
        :jobtracker  => { :run_state => :start, },
        :datanode    => { :run_state => :start, },
        :tasktracker => { :run_state => :stop,  },
      },
    })

end
```

Appendix D: Usage of KTHFS Portal and Virtualization

KTHFS Portal:

-We login into the portal with the user credentials.

-We select the provider and will load the specific form for the provider:

- We fill the rest of the form as follows:
    - We fill our cloud provider credentials account in the box provided (id and key).
    - We fill the common configuration parameters as follows:
        - Hardware ID: m1.medium or m1.large
        - Image ID: use eu-west-1/ami-ffcdce8b , this is the only one it has been working for now it is an Ubuntu 12.04 desktop version
        - Location ID: use eu-west-1
        - Image Login user: Do not tick and do not fill the login user field, this is for OpenStack. EC2 will not use it.
    - We add our public key for node management.
    - We click the launch button
    - We see a progress message bar appearing and writing messages.

-If everything ran fine, a message appears with the URL to the dashboard. Copy and paste in browser tab to go to the newly created dashboard in your cloud provider.

Virtualization:

In our newly created dashboard we login with our user (same as portal). Once we are in the Hadoop dashboard, we need to first setup the login credentials.

-To setup the credentials, click in the user icon on the left. A menu pops up, select setup credentials.

-We appear in a similar page to the one used by the portal.

- We click on the enable OpenStack button to disable that option.
- We fill the credentials for EC2 (like in the portal)
- We add our public key for node management of the cluster.
- Press save, a message will appear confirming

-We then go to the "add cluster" button and press it

-We upload an .yml cluster, in the dashboard code for the system to deploy.

-Once we upload, we see a confirm page, press next.

-We appear in a new page; here we see information of all the nodes to be launched plus information from the cluster to launch.

-To start the process, press start cluster. The process will start and it will take long time to complete

-Once it finishes, probably the login session will have timed out due to inactivity, login again and the hosts should appear in the host’s page. Navigate to check what is up or not in the dashboard options.

References:

1. D'Souza, J. C. (2013). *KTHFS–A HIGHLY AVAILABLE AND SCALABLE FILE SYSTEM*. KTH. http://kth.diva-portal.org/smash/get/diva2:603878/FULLTEXT01

2. Wasif, Malik. 2012. A Distributed Namespace for a Distributed File System. KTH. http://kth.diva-portal.org/smash/get/diva2:548037/FULLTEXT01

3. Shvachko, K., Kuang, H., Radia, S., & Chansler, R. (2010). The hadoop distributed file system. *Mass Storage Systems and Technologies (MSST), 2010 IEEE 26th Symposium on*. IEEE.

4. Mell, P., & Grance, T. (2011). The NIST definition of cloud computing. *NIST special publication*, *800*, 145.

5. Amazon. About Amazon Web Services. http://aws.amazon.com/about-aws/ [Online, Last Accessed: June 4, 2013]

6. OpenStack Foundation. OpenStack Software. http://www.openstack.org/software/ [Online, Last Accessed: June 4, 2013]

7. Schaller, R. R. (1997). Moore's law: past, present and future. *Spectrum, IEEE*, *34*(6), 52-59.

8. Bart Czernick. IaaS, PaaS and SaaS Terms clearly explained and defined. http://www.silverlighthack.com/post/2011/02/27/IaaS-PaaS-and-SaaS-Terms-Explained-and-Defined.aspx [Online, Published: February 27, 2011 Last Accessed: June 04, 2013]

9. Bunch, C., Arora, V., Chohan, N., Krintz, C., Hegde, S., & Srivastava, A. (2012). A Pluggable Autoscaling Service for Open Cloud PaaS Systems. *Proceedings of the 2012 IEEE/ACM Fifth International Conference on Utility and Cloud Computing*. IEEE Computer Society.

10. Steve Loughran, Hadoop in Cloud Infrastructures. http://steveloughran.blogspot.ro/2012/03/hadoop-in-cloud-infrastructures.html [Online, Published: March 3, 2012. Last Accessed: June 4, 2013]

11. VMWare, Serengeti Project. Virtualizing Apache Hadoop, Technical Whitepaper. http://serengeti.cloudfoundry.com/pdf/Virtualizing-Apache-Hadoop.pdf

12. VMWare & Hortonworks. Apache Hadoop 1.0 High Availability Solution on VMware vSphere ™ Technical White Paper v 1.0 http://www.vmware.com/files/pdf/Apache-Hadoop-VMware-HA-solution.pdf

13. VMWare, Serengeti Project. Hadoop Virtualization Extensions on VMWare vSphere 5 Technical White Paper. http://serengeti.cloudfoundry.com/pdf/Hadoop%20Virtualization%20Extensions%20on%20VMware%20vSphere%205.pdf

14. VMWare, Serengeti Project. Serengeti User's Guide v 0.8. http://serengeti.cloudfoundry.com/pdf/Serengeti-Users-Guide.pdf

15. Crockford, Douglas. "Rfc4627: Javascript object notation." 2006.

16. Hortonworks, Hortonworks Data Platform. Apache Hadoop Big Data Refinery White Paper. http://hortonworks.com/wp-content/uploads/2012/06/Apache-Hadoop-Big-Data-Refinery-WP.pdf

17. Hortonworks, Hortonworks Data Platform. Hortonworks Data Platform DataSheet. http://hortonworks.com/wp-content/uploads/downloads/2013/01/HortonworksDatasheet.HDP1_.2.pdf

18. Puppet labs, Puppet http://puppetlabs.com/puppet/what-is-puppet/ [Online, Last Accessed: June 24, 2013]

19. Ganglia, Ganglia Monitoring System http://ganglia.sourceforge.net/ [Online, Last Accessed: June 24, 2013]

20. Nagios, Nagios Monitoring System http://www.nagios.org/ [Online, Last Accessed: June 24, 2013]

21. Hortonworks, Apache Ambari Project. Ambari Architecture. https://issues.apache.org/jira/secure/attachment/12559939/Ambari_Architecture.pdf

22. Ronstrom, Mikael, and Lars Thalmann. "MySQL cluster architecture overview." *MySQL Technical White Paper* (2004).

23. Barb Darrow, Gigaom. Who's the biggest cloud of all? The numbers are in. http://gigaom.com/2013/03/11/whos-the-biggest-cloud-of-all-the-numbers-are-in/ [Online, Published: March 11, 2013. Last Accessed: June 5, 2013]

24. OpenStack, OpenStack Foundation. OpenStack Release Cycle. https://wiki.openstack.org/wiki/Release_Cycle [Online, Last Accessed: June 5, 2013]

25. Infochimps, Ironfan. Infochimps-cloud, tools Ironfan. http://www.infochimps.com/infochimps-cloud/tools/ironfan/ [Online, Last Accessed: June 5, 2013]

26. Opscode, Chef. Chef Overview. http://docs.opscode.com/chef_overview.html [Online, Last Accessed: June 5, 2013]

27. Amazon, Amazon OpsWorks. http://aws.amazon.com/opsworks/ [Online, Last Accessed: June 24, 2013]

28. Hashicorp, Vagrantup. Vagrantup, Why Vagrant? http://docs.vagrantup.com/v2/why-vagrant/index.html [Online, Last Accessed: June 5, 2013]

29. Cloudify, Cloudify community. http://www.cloudifysource.org/ [Online, Last Accessed: June 24, 2013]

30. Heroku, Heroku cloud platform. https://www.heroku.com/ [Online, Last Accessed: June 24, 2013]

31. Google, Google app engine platform. https://developers.google.com/appengine/ [Online, Last Accessed: June 24, 2013]

32. Ironfan, Github. Ironfan-Pantry Hadoop cluster example. https://github.com/infochimps-labs/ironfan-pantry/blob/master/cookbooks/hadoop_cluster/example/clusters/bigass.rb [Online, Last Accessed: June 5, 2013]

33. Flanagan, David, and Yukihiro Matsumoto. *The ruby programming language*. O'Reilly Media, 2008.

34. Apache Incubator. Jclouds: MultiCloud Library. http://jclouds.incubator.apache.org/documentation/gettingstarted/what-is-jclouds/ [Online, Last Accessed: June 24, 2013]

35. Clojure, Clojure programming language. http://clojure.org/ [Online, Last Accessed: June 24, 2013]

36. Adrian Cole, Jclouds. High Level Overview of Jclouds. http://www.slideshare.net/phymata/jclouds-high-level-overview-by-adrian-cole [Online, Last Accessed: June 5, 2013]

37. Bray, Tim et al. "Extensible markup language (XML)." *World Wide Web Journal* 2.4 (1997): 27-66.Appendix:

38. Ben-Kiki, Oren, Clark Evans, and Brian Ingerson. "YAML Ain't Markup Language (YAML™) Version 1.2." *3rd Edition, Patched at 2009-10-01*.

39. SnakeYAML, YAML parser and emitter for Java. https://code.google.com/p/snakeyaml/ [Online, Last Accessed: June 6, 2013]

40. Joshua Bloch. How to design a good API and why it matters. http://lcsd05.cs.tamu.edu/slides/keynote.pdf [Online, Last Accessed: June 6, 2013]

41. Parr, Terence. "The Definitive ANTLR Reference: Building Domain-Specific Languages (Pragmatic Programmers)." *Pragmatic Bookshelf, May* (2007).

42. Varaksin, Oleg, and Mert Caliskan. *PrimeFaces Cookbook*. Packt Publishing Ltd, 2013.

43. PrimeFaces, Primefaces JSF Suite http://primefaces.org/ [Online, Last Accessed: June 06, 2013]

44. Shaw, Mary, and David Garlan. "Software architecture: perspectives on an emerging discipline." (1996).

45. Gamma, Erich et al. *Design patterns: Abstraction and reuse of object-oriented design*. Springer Berlin Heidelberg, 2001.

46. Chatterjee, Arunodaya. "Futures: a mechanism for concurrency among objects." *Proceedings of the 1989 ACM/IEEE conference on Supercomputing* 1 Aug. 1989: 562-567.

47. Google. Google Guava Library. https://code.google.com/p/guava-libraries/ [Online, Last Accessed: June 6, 2013]

48. Tomasz Nurkiewicz. Listenable Futures in Guava. http://nurkiewicz.blogspot.se/2013/02/listenablefuture-in-guava.html [Online, Published: February 25, 2013. Last Accessed: June 6, 2013]

49. Tomasz Nurkiewicz. Advanced Listenable Futures. http://nurkiewicz.blogspot.se/2013/02/advanced-listenablefuture-capabilities.html [Online, Published: February 28, 2013. Last Accessed: June 6, 2013]

50. Dean, Jeffrey, and Sanjay Ghemawat. "MapReduce: simplified data processing on large clusters." *Communications of the ACM* 51.1 (2008): 107-113.

51. Java's Cyclic Barrier Abstraction. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/CyclicBarrier.html [Online, Last Accessed: June 6, 2013]

52. Java's CountDownLatch Abstraction. http://docs.oracle.com/javase/6/docs/api/java/util/concurrent/CountDownLatch.html [Online, Last Accessed: June 6, 2013]

53. Daniel Berninger, What the heck is an ECU? http://cloudpricecalculator.com/blog/hello-world/ [Online, Published: October 18, 2010. Last Accessed: June 30, 2013]