Automatic Discovery of Host Machines in Cloudify-powered Cluster

Lauri Suomalainen

Master’s Thesis
UNIVERSITY OF HELSINKI
Department of Computer Science

Helsinki, May 31, 2019

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty Faculty of Science	Laitos — Institution — Department Department of Computer Science

Tekijä — Författare — Author Lauri Suomalainen
Työn nimi — Arbetets titel — Title Automatic Discovery of Host Machines in Cloudify-powered Cluster
Oppiaine — Läroämne — Subject Computer Science

Työn laji — Arbetets art — Level Master's Thesis	Aika — Datum — Month and year May 31, 2019	Sivumäärä — Sidoantal — Number of pages 0

Tiivistelmä — Referat — Abstract

Hybrid Clouds are one of the most notable trends in the current cloud computing paradigm and bare-metal cloud computing is also gaining traction. This has created a demand for hybrid cloud management and abstraction tools. In this thesis I identify shortcomings in Cloudify's ability to handle generic bare-metal nodes. Cloudify is an open-source vendor agnostic hybrid cloud tool which allows using generic consumer-grade computers as cloud computing resources. It is not however capable to automatically manage joining and parting hosts in the cluster network nor does it retrieve any hardware data from the hosts, making the cluster management arduous and manual. I have designed and implemented a system which automates cluster creation and management and retrieves useful hardware data from hosts. I also perform experiments using the system which validate its correctness, usefulness and expandability.

Avainsanat — Nyckelord — Keywords
Virtualization, Distributed Systems, Containerization

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — Övriga uppgifter — Additional information

Contents

1 Introduction

Cloud adoption is growing ever so fast with vast majority of both enterprises and small and medium businesses leveraging on cloud computing one way or another [41]. While private cloud usage is growing at steady pace, its growth is eclipsed by that of public cloud usage which is estimated to grow trice as fast when compared to private clouds. Contributing to the accelerated speed of cloud adoption is the trend of simultaneous use of multiple cloud environments and services, both private and public. The concept of using multiple clouds to support and enable same business is called *Hybrid cloud* and on average enterprises report using and experimenting with almost five different clouds simultaneously. Another trend of cloud computing is a shift away from virtualised clouds to running workloads directly on hardware. This *bare-metal computing* interests companies running computationally heavy workloads such as Big Data and Machine learning as bare-metal seeks to amend performance overheads inherent to virtualisation. OpenStack Foundation reports a stark increase in the usage of its bare-metal service *Ironic* [43] and along with the possibility to use bare-metal servers with major public cloud providers there are also relatively new service providers such as *Vultr* [31] and *Packet* [20] who focus especially on providing bare-metal servers as a service.

Growing usage of both hybrid clouds and the variety of the underlying hardware and interfaces to use them introduce complexity to management of these systems. As a natural reaction, there are now many tools to abstract and manage this complexity. For example, IBM has their own tool *IBM Multicloud Manager* [11] and *Rancher* [22] has been a popular framework for handling multiple Kubernetes clusters [14]. This thesis focuses on *Cloudify* [3] which is also a tool to manage multiple clouds. What sets it apart from others however is the fact that it aims to be a general tool independent of the underlying platform implementations meaning that the user can control multiple clouds and even single physical machines as a generic set of resources without extensive knowledge of their implementation. This opens up avenues in optimising cloud resource usage and introducing hardware that has not traditionally been used as cloud computing resources such as consumer-grade computers and single-board computers such as Raspberry PIs. However, as bare-metal cloud computing is not as popular as applications of virtualised computing resources, Cloudify's bare-metal capabilities remain underdeveloped.

In this thesis I identify shortcomings related to Cloudify's capability of managing generic computational resources such, as consumer-grade computers, and provide prototypical solutions addressing them. Main problems addressed are Cloudify's inability to automatically detect and manage physical hosts in the cluster and its lacking knowledge of the performance capabilities of the said hosts. My key contributions are:

1. A software solution which detects joining and parting hosts in the cluster network automatically without a need for human intervention and provides them to the Cloudify Manager for allocation.

2. A modification to Cloudify's Host-pool service so that it retrieves and stores hardware data and performance capabilities of the hosts. In the future Cloudify Manager can use this data to optimise resource usage and make more intelligent workload allocation choices.

Both of the solutions integrate seamlessly with the existing Cloudify components. I also perform experiments on real machines to showcase and validate the capabilities and correctness of my solutions within the scope of this thesis. The features I am addressing are lacking likely because Cloudify's development team's focus has been on integrations with the major cloud platforms and generic hardware provisioning is a niche use case compared to them.

The remainder of this thesis is structured as follows: First in section 2 I give a background overview of common cloud computing concepts. Then I follow with the background review of Cloudify, comparing it conceptually to OpenStack which serves as an example of a typical Cloud computing platform. I also provide a quick overview of hybrid cloud and bare-metal management tools similar to Cloudify. From section 3 onwards I focus on identifying the scope of the prototype and the shortcomings of Cloudify I set out to correct. I provide an overview of the parts in Cloudify with which my proposed system interacts with and detail a high level design of my solutions for automating host detection and retrieval and storage of hardware data. Section 4 presents the lower level details of solutions' implementation followed by the experiments in section 5 showcasing and validating the solutions' capabilities. Finally in chapter 6 I review future work and research required to fully develop the system beyond the prototype.

Both solutions, Discovery Service and Modified Host-pool Service, presented in this thesis are open source.[1]

[1]The Discovery Service is available at `https://bitbucket.org/Fleuri/discoveryserviceforcloudify/src/master/`. The modified Host-pool Service is available at `https://github.com/Fleuri/cloudify-host-pool-service`.

2 Background

Often heard quote about cloud computing is that *"There is no cloud. It is just someone else's computer"*, implying that cloud computing is just traditional distributed computing marketed with a more attractive name. While the core of the cloud is undeniably in distributed computing, cloud computing as a whole can be seen as a fundamental paradigm shift in which the hardware and software is abstracted to the end user and the resources are offered as different types of services. [38]

In cloud computing, there are multiple recognised service models which dictate how the users can use the given system and what privileges they are given [48]. In its most limited form, a cloud service is offered to a user as a predefined application or a set of applications. The user has some interface for interacting with the applications but is given no control over anything else such as other applications, the operating system the application is running on or network and hardware configurations. This is generally known as Software as a Service (SaaS). The most permissible service model is known as IaaS, Infrastructure as a Service. In its archetype the user gets access to all fundamental computing resources, possibly including some network components, and can run arbitrary software including operating systems. The user experience should be similar to that with their personal computers. The user is not allowed to access the underlying cloud infrastructure. Between the two falls Platform as a Service (PaaS). PaaS typically allows users to deploy their own applications along with their dependent libraries, tools, services etc. provided that they are supported by the cloud provider. The user has no control over underlying cloud infrastructure, operating system, storage or network but usually can configure certain settings and possibly choose different supporting services the cloud provider offers. There are also other "aaS" such as Data as a Service (DaaS) and Storage as a Service (SaaS) but they are based on one of the three aforementioned service models or are variations or subsets of them. Sometimes the numerous models are referenced with umbrella terms of XaaS and EaaS meaning Everything as a Service for both or Anything as a Service for the former [38].

2.1 Virtualisation

Virtualisation in the context of distributed cloud environments usually refers to virtual machines. The core idea is analogous to computer hardware virtualisation. Operating systems offer an interface for the processes to utilise the computer hardware while giving them an illusion that they have all of the hardware for themselves [34]. In reality the resources are shared among many processes. Likewise in cloud environments resources are being share by processes but also by different users running different operating systems, configurations and programs. As with the processes, users are given an impression that they alone have access to the underlying hardware resources, whereas in reality there are multiple users using the same physical machines.

There are several reasons as to why would one prefer a virtualised environment to a non-virtualised one.

1. Hardware utilisation

Obviously in multitenant cloud services it is crucial for the service provider to maximise the use of their hardware resources. Thus it is imperative for the provider to try to share the limited hardware resources among as many users' virtualised environments as possible. Otherwise every user would need their own physical machine in the system which would both require more resources per user and leave resources underused. For example a 2018 study showed that even a typical public computing cluster uses around half of the CPU and memory resources available to it. [53]

2. Fault tolerance

From the fault tolerance perspective, using virtual machines in a distributed environment decreases their dependency on the underlying physical hardware [36]. That is because in virtual machine architectures which support live migration of operating system instances can be seamlessly moved from one physical machine to another. This also helps the load-balancing in the distributed system and allows low-level and physical maintenance of the hardware without considerably disrupting the usage of the system.

3. Flexibility

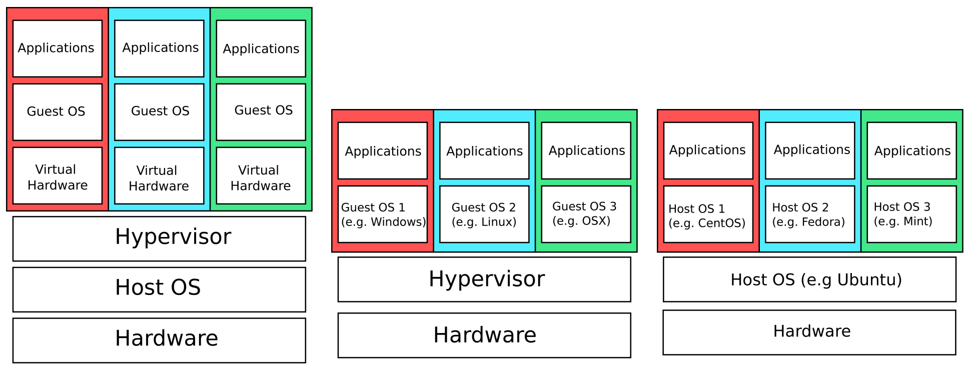
An end-user also has many reasons to use virtualised cloud services. User only needs a lightweight computer with an internet connection to perform computationally challenging tasks in the cloud back-end. Similarly devices with little storage capacity can leverage from a cloud service's vast storage space. Some users would like to use applications and programs not native to their operating system of choice making another virtualised OS a convenient option [34]. Virtualised environment allows software developers to test and debug their software with many different settings, as virtualised environments can have different operating systems and available hardware resources. Naturally this also allows emulating completely different devices [39].

2.2 Heterogeneous clouds, bare-metal and hybrid

One of the most common assumptions of the current cloud computing paradigm is that the cloud environment is built on commodity hardware [37]. Even if that was not the case, virtualisation and orchestration techniques typically abstract the underlying hardware making it invisible to users. This can cause problems if the cluster's devices' capabilities differ from each other drastically. In a multitenant cloud the use cases, workloads and resource needs differ between users, but the cloud is only capable of offering generic solutions for everyone.

Other motivations to deploy heterogeneous hardware to data centres relating to different use cases and needs stem from bare-metal solutions and green computing movement [45]. For example, if a user is running mainly computationally light applications that perhaps only run for a short time, it is wasteful to keep full-fledged rack servers running if the same task could be accomplished with hardware requiring less power and outputting less heat e.g. a Raspberry Pi [23]. In addition, such machines are magnitudes cheaper than traditional rack servers. Virtualisation techniques deployed in current clouds have wide range of benefits but they incur overheads making them undesirable for certain high-performance computing tasks [46]. Such tasks may also require specialised hardware to optimise the performance and thus in the best scenario user should have information of the hardware capabilities and be able to control on which nodes their tasks are run.

Ability to know and control nodes and their capabilities are also relevant hybrid clouds. One way to classify different clouds is by which party offers the service. Clouds hosted by an organisation meant for its internal use are referred to as *Private clouds* whereas cloud service offered by an organisation for other party to rent and use is known as *Public clouds* [42]. Hybrid cloud is typically combination of these two, but could also refer to any separate cloud platforms used together. An organisation may need to provision resources from a public cloud occasionally for different use-cases and workloads to complement their own environment or the private cloud is used to control data more securely as the service using the data is offered in a public cloud. Use cases and motivations for deploying a hybrid cloud vary, but a result is most likely a cluster with heterogeneous hardware.



(a) Full (b) Hardware-Layer (c) Container-based

Figure 1: **Popular virtualisation techniques**. *Along with a) full virtualisation, b) hardware-layer virtualisation, and c) container-based virtualisation, other virtualisation techniques include unikernels and paravirtualisation. Bare-metal computing which gives the users complete control over the computing resources is also getting popular.*

2.3 Virtualisation Techniques

Traditionally virtualisation has referred to a software abstraction layer residing between the computer hardware and the operating system [51]. This layer has been called Virtual Machine Monitor (VMM) or more recently a hypervisor and it hides and abstracts the computing resources from the OS, allowing multiple OSes to run simultaneously on the same hardware. There are multiple ways to run hypervisor-based virtualisation. Lately a technology called container-based virtualisation has been gaining popularity. Instead of emulating whole hardware, containers make use of features provided by the host operating system to isolate processes from each other and other containers [39]. Cloud computing in which the host machines are not virtualised is known as bare-metal computing [46].

2.3.1 Full virtualisation

In full virtualisation, the hypervisor runs on top of the host OS. The guest OSes run on top of the hypervisor which in turn emulates the underlying real hardware to them. The hypervisors running on top of the host OS are generally referred as *Type 2 Hypervisors* [39]. The guest OSes can be arbitrary. Figure 1a shows the full virtualisation architecture with the hypervisor running on top of the Host OS and Guest OSes on top of the hypervisor using their emulated hardware.

The main advantage of full virtualisation is that it is easy to deploy and should not pose problems to an average user but the virtualisation overhead results in significantly reduced performance when compared to running directly on hardware [51]. Popular examples of full virtualisation applications are Oracle's *VirtualBox*[19] and *VMware Workstation*[30].

2.3.2 Hardware-Layer virtualisation

Hardware-Layer virtualisation is also a type of full virtualisation, but unlike Type 2 hypervisors, the so called *Type 1 Hypervisors* (also *native* and *bare metal*) run directly on hardware. As seen in figure 1b, there's no Host OS per se. Instead the Guest OSes access to hardware resources is controlled by the hypervisor.

Running directly on hardware, Hardware-Layer virtualisation techniques suffer less performance overhead than their OS-layer counterparts [51]. On the other hand, Type 2 hypervisors being essentially applications themselves can be ran in parallel on the host OS whereas Type 1 hypervisors can not. For an average user, setting up a Type 1 hypervisor can be more difficult than Type 2. Commercial examples of Type 1 Hypervisors include Microsoft's *Hyper-V*[15] and VMware's *VSphere* [29].

2.3.3 Container-based virtualisation

Instead of virtualising the underlying hardware, container-based virtualisation also known as OS-Layer virtualisation [51] focuses on user space and allows running multiple operating systems in parallel as applications using the same kernel as the host operating system. A prime example of a popular container-based virtualisation platform is *Docker* [8] which leverages on native Linux kernel features to virtualise and isolate OS instances. Figure 1c shows a container-based virtualisation architecture in which containerised environments are running operating systems on host OS's kernel.

Container-based virtualisation does not need to emulate the hardware as containers communicate directly with the host kernel [39] and are thus very fast to start. They also do not require all of the components a fully virtualised environment would need to run and therefore their resource fingerprint is minimal when compared to hypervisor-based virtualisation techniques.

The obvious drawback of the technique is that the kernel of the virtualised OS has to be the same as that of Host OS e.g. In a situation depicted in figure 1c operating systems based on Linux kernel could be ran on Ubuntu Host OS but OSes like Windows or OSX could not. On certain virtualisation platforms resource-intensive containers can also affect other containers detrimentally as the shared host OS's kernel is forced to spend its execution time on handling the instructions from the stressed container [54].

2.3.4 Paravirtualisation

Paravirtualisation differs from full virtualisation by requiring the Guest OS to be modified in order to accommodate the virtual environment in which it is ran. Otherwise the architecture is similar to that of full virtualisation, but with thinner hypervisor allowing performance close to that of a non-virtualised environment. A well-known example of a paravirtualisation hypervisor is *Xen* [32].

2.3.5 Unikernels

Unikernels are a relatively recent take on virtualising services. Building on the notion that in cloud environments each VM usually specialises to provide only one service even if each VM contains a full-fledged general computer [47]. Unikernels are essentially minimal single-purpose library operating system (*LibOS*)[49] VMs with a single address space. They contain only the minimal set of services, implemented as libraries, built and sealed against modification to run the one application. Unlike the earlier LibOSes unikernels do not require a host OS to run but run directly on a VM hypervisor, such as Xen.

Some benefits of unikernels are obvious. Constructing VMs with minimal set of service libraries results in small images and resource footprints as well as fast boot times. Other benefits include reduced attack surface due to smaller code base and sealing preventing any code not compiled during the creation of the VM from running. Single-address space improves context switching and eliminates the need for privilege transitions making system calls as efficient as function calls [44]. Running directly on the hypervisor instead of a host OS eliminates superfluous levels of hardware abstraction.

Optimisation and simplification are not without drawbacks. By definition, unikernels are not intended for general-purpose multi-user computing but for microservice cloud environments. Running multiple applications on a single VM is risky due single-address space does not offer any inherent resource isolation. As unikernels are sealed during compiling, it is not possible to do changes to them afterwards. User is instead required to compile and launch a completely new modified VM.

Popular examples of unikernels are *MirageOS*[16] and *OSv*[44].

2.3.6 Bare-metal cloud computing

While virtualisation is often desirable for its flexibility, multi-tenancy and other attributes, there are use cases in the cloud where a user would rather forego virtualisation. Bare-metal cloud computing refers to a practice of running distributed workloads directly on cloud's physical servers much like one would with virtualised servers: Similar elements include for example abstraction and on-demand provisioning. Bare-metal is often preferred in High Powered Computing (HPC) use cases for maximum utilisation of computing power. Bare-metal's benefits include non-existent virtualisation overhead, ability to choose the hardware the workload runs on and can tune it for maximum performance, and single-tenancy ensuring that no other users are running workloads on the same physical machine which could interfere with each other [50]. On the other hand the aforementioned flexibility is lost and single-tenancy poses challenges on workloads if maximum resource usage is desired.

Prominent bare-metal provisioning platforms include OpenStack Ironic [12], Canonical Maas [1] and Razor [24].

2.4 Cloudify and Cloud Management Platforms

Enterprises are using increasingly more distinct clouds simultaneously [41] and the clouds themselves are becoming bigger and more complex. Different clouds have different features and capabilities, are used differently and are not always interoperable [40]. This has created demand for tools to manage the scale and complexity of these systems. These range from integration libraries like *jclouds* [13] to full-fledged management frameworks like IBM Multicloud Manager, Cloud Foundry and Cloudify [11, 2, 3] which offer unified resource abstraction, orchestration and deployment capabilities among others.

In the following sections I provide background to Cloudify and motivate its use in this thesis. I also discuss OpenStack as an example of a typical Cloud Platform and compare it to Cloudify to point out the differences and similarities between them and other cloud platforms and cloud management platforms in general.

2.4.1 Cloudify

Cloudify [3] is an open-source orchestration software aiming to provide a unified control and modelling layer for common cloud computing platforms. Cloudify can be used to uniformly orchestrate heterogeneous sets of both virtual and physical cloud resources such as networking, computing and storage resources and even pieces of software. They can also be provided from different environments such as OpenStack, AWS, Google Cloud Platform (GCP), Kubernetes and even bare-metal clouds. Orchestrating different versions of the same underlying cloud environment is also possible. The applications, workflows and the cloud infrastructure itself is described with OASIS TOSCA [33] based Domain Specific Language (DSL) in configuration files called *blueprints* in Cloudify jargon. Configuration files are vendor-agnostic, meaning the same configuration can be reused with different underlying infrastructure. Cloudify *plugins* act as an abstraction layer between the generic blueprints and cloud environments' more specialised APIs. The generalising approach makes Cloudify suitable for hybrid clouds and allows seamless migration of resources between different environments.

2.4.2 OpenStack

OpenStack [17] is an open-source software platform for cloud computing. A project originally founded by NASA and Rackspace Inc. now has a large base of supporting companies [7] and a thriving community. OpenStack allows its users to deploy a full-fledged cloud computing infrastructure. User can control pools of both physical and virtual computing, storage and networking resources. It can be run on commodity hardware and supports a plethora of enterprise- and open source technologies making it possible to use heterogeneous physical and software environments. OpenStack consists of different projects that provide services for the system. A user can freely choose which services to deploy. Project range from essential *Core Services* like computing, block storage, identity service and networking to more specific and specialised such as MapReduce and Bare-metal provisioning[17]. OpenStack boasts many features: It is massively scalable supporting up to million physical and 80 million virtual machines [52]. It also supports a wide array of market-leading virtualisation technologies such as QEMU, KVM and Xen and it is fully open-source with thriving community and industry backing [7]. Other features include fine-grained access control and multi-tenancy, fault-tolerance and self-healing [18].

2.4.3 Comparison of OpenStack and other Cloud Management Platforms

Both OpenStack and Cloudify are used to operate a large number of computing, networking and storage resources. However, they are not directly comparable. While Cloudify can be used to orchestrate resources and applications on a cloud platform, OpenStack is a cloud platform.

Similar orchestration project within OpenStack is Heat [10], which can be used similarly to Cloudify's DSL to write human-readable templates (*HOTs – Heat Orchestration Templates* as they are called in the Heat project) to automate deployments of applications and cloud resources. Heat orchestration is of course limited to OpenStack itself. Even though there are drivers which allow OpenStack to manage resources from major public clouds such as AWS and GCP (and thus allowing a public/private hybrid cloud), the resources are abstracted to those common to OpenStack: Heat cannot orchestrate them independently of an OpenStack deployment. Cloudify however is cloud-platform agnostic and it can manage multiple different cloud environments simultaneously, including OpenStack. On the subject of hybrid clouds, Cloudify supports bare-metal deployments by default and OpenStack's project *Ironic* for provisioning bare-metal instances has been integrated to OpenStack since 'Kilo' development cycle. Both Cloudify and OpenStack are open-source projects with notable contributing community but OpenStack has more industrial partners than Cloudify.

What makes Cloudify stand out however is its broadness, generality and expandability. Other frameworks like Cloud Foundry focus on common application stacks and mechanisms to streamline application development and deployment work while Cloudify allows user to orchestrate complex workflows on practically any platform, starting from infrastructure management ending as low as a single BASH script [4]. Among these capabilities is the ability to provision generic host machines as cloud resources without using a cloud platform. There are other systems which can provision generic hosts similarly, such as Red Hat Satellite [25] or Docker Machine [9] but unlike them Cloudify does not require installation of any additional software on the hosts. Additionally preparing the hosts for provisioning seems to require human intervention in most cases, including Cloudify. Simplifying and automating this task as well as providing more insight to the hosts' capabilities are the main focuses of this thesis.

3 System Design and Implementation

Current cloud management platforms make simplified assumptions about the hardware in the datacentre and its usage. Hardware is by default powerful rack or blade servers, they are virtualised, and always on. Thus cloud management platforms on the market are sub-optimal for certain use cases.

HPC and Big Data applications require highly optimised and powerful hardware. In such applications, the overheads imposed by virtualisation are undesirable and for maximum efficiency the cluster should consist of bare-metal computing nodes. Furthermore other advantages of virtualisation such as multi-tenancy and scaling are not useful in bare-metal computing.

On the other end of the spectrum are very weak computers with limited computing power, memory, I/O throughput and storage. These machines can be a worthwhile addition to a cloud environment for running small low intensity tasks: They are significantly cheaper to traditional datacentre hardware costing some hundred Euros per machine instead of thousands like a single rack server. They do not require much space to store, use less electricity and output less heat. Virtualisation may not be applicable for such machines either due to hardware not supporting virtualisation in the first place, or as the virtualisation overhead may consume large enough share of a machine's resources rendering it incapable to perform or at least severely restricting any other functionality besides virtualisation. With low end computers virtualisation benefits like multi-tenancy and running multiple operating systems in parallel may simply not be possible because of limited capabilities. Using these machines in a heterogeneous cluster requires treating them like a traditional bare-metal nodes, albeit not nearly as powerful.

In order to leverage on bare-metal nodes in a heterogeneous and possibly even in a hybrid cloud, the task schedulers require a view to the underlying infrastructure so that they can allocate tasks to nodes fitted to perform them. To extend the usage to befit hybrid clouds in addition to heterogeneous, the orchestrator has to be vendor agnostic too. This thesis presents prototype extensions to Cloudify's [3] client agents, which are used to communicate between the nodes and Cloudify Manager. Extensions are going to allow two things:

1. *Allow the Manager to gain information about the nodes' hardware capabilities, a feature that is currently lacking from the project.*

2. *Enable node discovery in the cluster.*

Currently managing the composition of the host-pool is a manual effort. The hosts nodes in the cluster can either be configured before launching the cluster using a host pool YAML file or with REST API calls. Therefore monitoring for failing nodes and adding new ones, especially en masse, is an arduous task. A discovery mechanism for new nodes in the cluster would ameliorate if not solve the problem, even if replacing faulty hardware is more often than not a manual task. Additionally discovery mechanism would allow *Bring-your-own-host* kind of functionality.

3.1 Design overview

Centrepiece in Cloudify's architecture for achieving the set goals for more detailed information and node discovery is the *Host-pool service* [5]. Host-pool service is a RESTful service to which Cloudify Manager can make calls via Host-pool plugin to gain information about nodes that compose the cluster. It can also allocate hosts for jobs run by the manager as well as deallocate them. One major feature host-pool service provides is adding hosts to the pool during runtime. It can also remove hosts from the pool and both operations are performed with a similar REST API call. A Cloudify set-up without host-pool service can not make use of generic cluster comprising of different bare-metal nodes. The relationships between different Cloudify components are illustrated in figure 2.

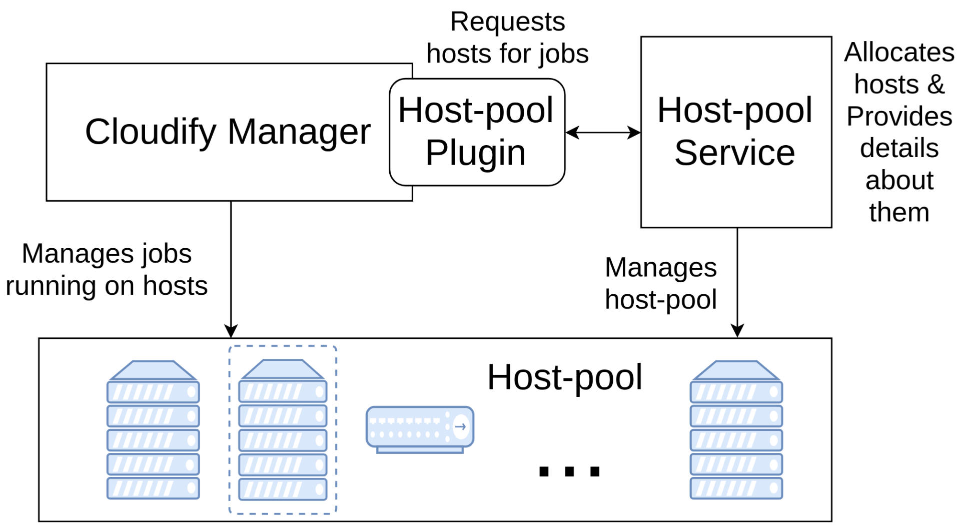


Figure 2: The role of the Host-Pool Service.

The goal for retrieving more hardware information from hosts requires running a script on the hosts. Currently the information host-pool service provides is concise, providing information like the operating system of the host, available endpoints and login credentials. It does not in fact query the host themselves. In order to get any performance information, host-pool service should run a script querying for the hardware data and storing the results when adding the host to the logical pool. This requires extending the host-pool service.

The goal for host discovery and automated host pool management will be implemented as an additional Discovery Service. The role of the Discovery service is twofold as seen on figure 3.

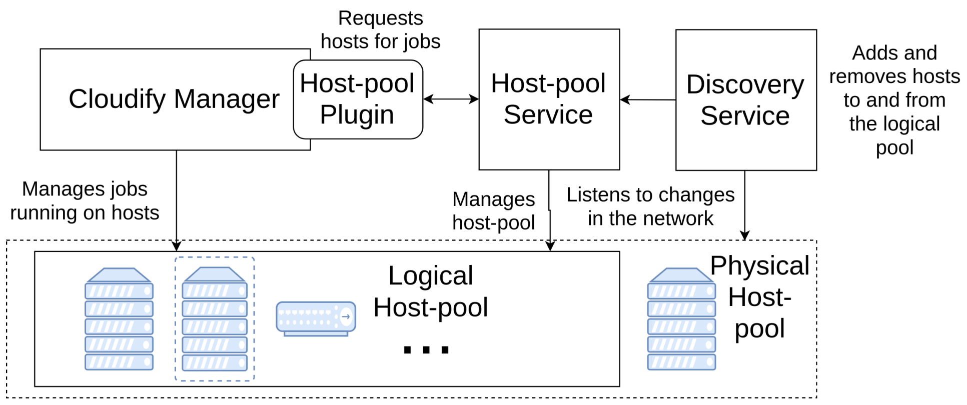


Figure 3: Discovery service's relation to other Cloudify Components.

The Discovery Service constantly monitors the network and its devices and keeps track of discovered devices and their health. When a new device is detected in the network its details are added to Discovery Service's local memory and a REST API call to add the device to the logical pool of hosts is made to Host-pool Service. Device health is monitored periodically and after a set number of failed health checks are detected, the device is removed from Discovery Service's memory and a REST API call to remove the host from the logical pool is made.

4 Technical Implementation

Discovery Service is implemented with Python programming language and Flask Web framework. They were chosen because all of the components in Cloudify are also written in Python and Flask is used primarily for providing REST APIs. Even though Discovery Service does not provide any REST APIs, Flask is used for configuration management and source code organisation. Naturally, if need arises in the future to expand Discovery Service with a REST API, the development work is streamlined because of the framework.

In addition to Python program, Discovery Service relies on Redis [26] as an in-memory key-value storage. Redis is a completely separate process in addition to Discovery Service. The preferred way of deploying Redis is in a docker container as it doesn't require installation or configuration save for exposing a correct port in the container and specifying Redis' address to Discovery Service. Redis could also be installed in the host system or even a remote system, though latter option has no practical purpose due to network latency as Redis achieves its high performance by storing values in the memory instead of disk.

On the source code level, Discovery Service consists of two major components: The network scanner and request service. The network scanner is given a subnet as a parameter and it constantly sniffs the network detecting joining and already present devices and keeping track of them. Its other task is to periodically send health checks to known devices and if a health check fails enough times, it removes the given device from the logical host pool. Request service is responsible for sending HTTP requests to the Cloudify Host-pool service. It is called by the Request Service and it runs asynchronously. In addition to HTTP requests, it performs checks to ensure that the state of Discovery service's and Host-pool service's databases correlate.

The communication relationships between different components in the system are depicted in figure 4. The source code for Discovery Service as well as the documentation can be found at `https://bitbucket.org/Fleuri/discoveryserviceforcloudify/src/master/`

4.1 Network Scanner

The Network Scanner is responsible of monitoring the network allocated for the Cloudify-orchestrated bare-metal cluster. It does so by passively listening to the network traffic but also by actively pinging the already discovered hosts. Network Scanner has two main functions, sniffer and pinger, and they run concurrently on two threads. Sniffer listens to ARP

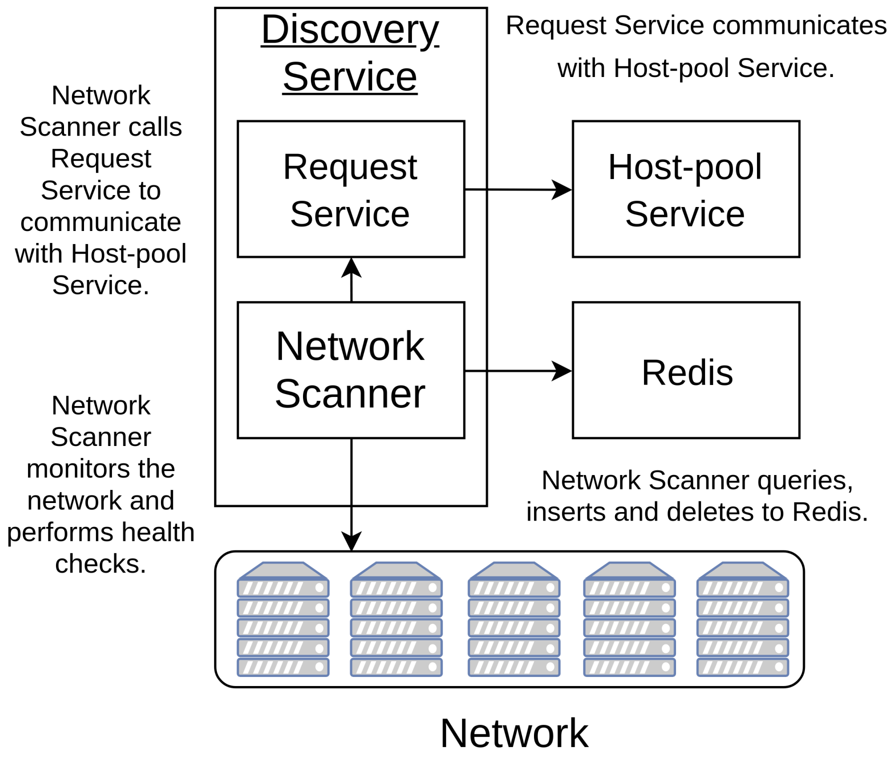

Discovery Service
Request Service communicates with Host-pool Service.
Network Scanner calls Request Service to communicate with Host-pool Service.
Request Service
Host-pool Service
Network Scanner
Redis
Network Scanner monitors the network and performs health checks.
Network Scanner queries, inserts and deletes to Redis.
Network

Figure 4: The communication relationships between Discovery Service and other components

packets in the network and upon receiving one, stores the details of the sender device. Pinger function periodically sends ARP pings to previously discovered device and keeps track of their successes, eventually removing unresponsive devices from the logical host pool. In addition to two main functions there's a start-up function that initialises both local Redis storage and the Host-pool service's database. It pings all of the IP addresses in the given IP range and stores the found device details to databases.

4.1.1 Sniffer

Sniffer is the part of the Network Scanner used to passively listen to the network traffic in the cluster's network, and detect and store joining hosts.

Sniffer is started in its own thread during the start up sequence of the Discovery Service. It uses *Scapy* library [27] for Python. Sniffer function is given three arguments:

1. **The network interface** which the Sniffer listens to for incoming packets.

2. **The callback function** which details further instructions to perform when a packet is caught.

3. **The filter** which restricts the type of packets caught.

Only the interface can be set by the user of the Discovery Service. The callback function is the core application logic of the sniffer and the Discovery Service itself relies on sniffing ARP packets and therefore the filter is set accordingly.

When it comes to programming logic, the callback function is the most interesting part of the sniffer. Its purpose is to evaluate whether an ARP request comes from a new or know device and store details about them. When the function receives an ARP packet it first filters out packets that are not standard ARP requests. There are two such cases: ARP Probe [35], in which the source IP address or hardware address of the ARP request is *0.0.0.0* or *00:00:00:00:00:00* respectively, and gratuitous ARP in which the hardware address is *ff:ff:ff:ff:ff:ff*. The user can also define a list of IP and hardware addresses which the Discovery Service should ignore i.e. devices on which Cloudify should not run workloads. Such devices include the host on which Cloudify manager runs and network devices such as routers.

Next the function checks whether the packet's origin is an already known host by querying Redis. If the host is not previously known or if it is a known device with a changed IP address, the function starts a new thread to add or patch the host to the Host-pool service. See section 4.2.

Whether the packet's origin host is known or not, the next step in the function is to insert values extracted from the packet to Redis. The data structure Discovery service uses is simple, being a hash table with the devices' hardware address being a key and value being a dictionary object consisting of the given device's IP address and the number of failed ping attempts. See section 4.1.3 for more details. If the packet's origin is a new host, its key and values are inserted to the data store with number of failed pings always being zero. If a device is already known, its hardware address is already stored to Redis and therefore its values are modified: in most cases only the failed ping count is reset to zero but there could be cases in which the device's IP address has changed and it is updated here accordingly. Algorithm 1 details the structure of the function. Note that both adding a new host and patching an already known host is done in the same Request Service function. See section 4.2.2 for details.

Algorithm 1: Sniffer Callback Function

```
Input: Packet
if Packet is an ARP packet then
    if Packet is not an ARP Probe or Gratuitous ARP then
        if Hardware address not found in Redis or IP address not
        found in Redis then
            RequestService.Register_a_new_host();
        end
        Redis.store(Packet.HardwareAddress: {ip_address:
        Packet.IPAddress, ping_timeouts: 0});
    end
end
```

4.1.2 Start-up routine

Related to the sniffer function, when initialising the Discovery service, a start-up routine is run. It has three functions:

- It flushes the Redis key-value store
- It empties the logical host-pool on the Host-pool Service.
- It scans every IP address in the cluster network, adding any device found to the logical host-pool.

The routine follows the steps outlined above. First a flushing call is made to Redis. Then, using the Request Service detailed in the section 4.2, the start-up routine retrieves the ids of the current hosts in the host-pool service delete the entries one by one.

Finally the routine sends an ARP ping to each hosts (Manually excluded hosts do not apply) and upon receiving a reply, stores the details of the host as described in the section 1. At the prototypical state, the network scan waits for every host to either reply or timeout, making the start-up routine slow if subnet's IP range is large. Parallelism and other possible future optimisations are discussed in section 6

4.1.3 Pinger

Pinger is the part of the Network scanner which is used to perform health checks on existing hosts in the network and subsequently remove them from storage were they to fail them a certain number of times.

Pinger function is started in its own thread in the Network Scanner. Its responsibility is to keep track of the health of the nodes in the network. If Pinger discovers an unreachable node, it is removed from both Redis' and Host-pool service's storage.

Pinger periodically works through a list of known hosts in the network sending an ARP ping to each host. Upon receiving a response it resets the corresponding host's ping time-out counter to zero. If Pinger does not receive a response, it increments the given hosts' ping time-out by one. If after this operation the time-out crosses the given ping time-out threshold, the host is assumed to have disconnected from the network and is removed from the storage. After Pinger has pinged every host in the network, the process waits for a given ping interval after which it restarts the process.

The user gives inputs two parameters in a configuration file for Pinger to use: *ping_timeout_threshold* and *ping_interval*. Ping_timeout_threshold specifies the maximum consecutive ping failures that can occur before a host is marked unreachable and removed from storage. Ping_interval is the duration in seconds of which Pinger waits after each round of pinging the network. The more detailed presentation of the function is presented in the algorithm 2.

Algorithm 2: Pinger Algorithm

```
while True do
    foreach host in Redis do
        timeouts = host.ping_timeouts;
        response = Ping(host);
        if response then
            Redis.patch(host, {ping_timeouts: 0});
        end
        else
            timeouts++;
            if timeouts >= ping_timeout_threshold then
                RequestService.delete(host);
                Redis.delete(host);
            end
            else
                Redis.patch(host, {ping_timeouts: timeouts};
            end
        end
    end
    Sleep(ping_interval);
end
```

4.2 Request Service

Request Service is responsible of communication between the Discovery Service and Host-pool Service. As seen on figure 4, it is called by Network Scanner and it makes requests to Host-pool Service. The REST API Host-pool Service provides is quite succinct but provides a typical CRUD interface for handling nodes in the network. The methods are as follows[2]

[GET] /hosts

GET request to /hosts returns a JSON list of hosts and their details. Also accepts certain filters.

[POST] /hosts

POST request to /hosts with a valid JSON array will add one or more hosts to Host-Pool service's storage See listing 1 for the JSON schema definition. Returns Id's of the new host or hosts.

[2] *Note: Paths are relative to Host-pool service base URL e.g. localhost:8081/hosts*

[GET] /host/id
: Returns details of a single host corresponding to the ID number.

[PATCH] /host/id
: PATH request allows updating specified fields of a host with the given ID.

[DELETE] /host/id
: Removes the host with given ID from Host-pool service's storage.

[POST] /host/allocate
: This API call allocates a host to be used by the Cloudify Manager.

[DELETE] /host/id/deallocate
: This returns a previously allocated host back to the host pool to be reallocated again.

Request Service interacts with all of the REST API endpoints except for allocation and deallocation endpoints which are interacted by Cloudify Manager's Host-pool plugin.

On source code level a typical Request Service function either makes an HTTP request to a certain endpoint, possibly with an ID corresponding to a host in the network, or constructs a JSON payload and sends that along with a POST request. Programmatic challenge in Request Service rises from the fact that the data model Host-pool service accepts is significantly more robust than that of the Discovery service, as can be seen on listing 1. If Discovery Service's data model as seen on listing 2 was a subset of Host-pool service's, handling data in Request Service would be trivial. However as Discovery Service's data model uses hardware addresses as unique identifiers whereas Host-Pool Service attaches a running number to each host. Hardware addresses are used as they are immutable in the cluster use-case and Discovery Service accounts for possibly changing IP addresses. However, Hardware addresses are by to Host-pool Service implicitly, as Discovery Service assigns each host's hardware address as a value for 'name' key.

```
hosts: {
        id: Integer
        name: String
        os: String
        endpoint {
                ip : String
                protocol: String
                port: Integer
        }
        credentials: {
                username: String
                password: String
                key: String
        }
        tags: Array
        allocated: Boolean
        alive: Boolean
}
NOTES:
- name is an arbitrary string, but Discovery Service assigns
    the host’s hardware address as the value of name.
- os can be either ’linux’ or ’windows’. Other values are
   invalid.
- endpoint.ip has to be a valid IP address range with CIDR
   notation. If a range is defined, Host-pool Service
   considers each unique IP address a single host.
- protocol can be either ’winrm-http’, "’winrm-https’ or ’
   ssh’, but Host-pool service does not explicitly force
   this.
- Host-pool Service manages id, allocated and alive fields.
   For User, all other fields except credentials.password,
   credentials.key and tags are obligatory.
- In addition to ’hosts’ key, Host-pool Service also accepts
    ’defaults’ key. ’defaults’ can contain the same keys as
   ’hosts’. If ’defaults’ is provided, its values are
   appended to each host which has respective undefined
   values. id, allocated and alive cannot be provided as
   defaults.
```

Listing 1: JSON schema accepted by the Host-pool Service for a single host

```
hwaddress: {ip_address: "string", ping_timeouts:  "integer"}
```

Listing 2: Discovery Service data format for a single host

4.2.1 Id checking

Due to differing identifiers, certain functions are implemented to keep the two data stores synchronised. A `get_id()` -function retrieves a JSON object of hosts as depicted in algorithm 3. Then it compares stored IP addresses for each host or until a match is found after which the ID is returned. If hardware address is passed as an argument, the function also compares it to *name* -field's value, as Discovery Service names hosts in Host-pool Service after their hardware addresses. Hardware address check is used when Network Scanner finds an already known host with a changed IP address and calls Request Service to patch it to Host-pool Service. The ID itself is used for REST API calls that are directed at a single host.

Algorithm 3: get_id -function compares stored IP addresses and optionally hardware addresses to find out the corresponding ID number in Host-pool Service.

```
Input: ip_address
Input: hardware_address = None
foreach host in RequestService.get_hosts() do
    if host.endpoint.ip is ip_address or name is not None and
    host.name is hardware_address then
        return host.id;
    end
end
return None;
```

4.2.2 Adding a new host to Host-pool Service

When Network Scanner detects a new host, it adds it to Redis key-value storage and starts a Request Service routine `register_a_new_host` depicted in algorithm 4 in a new thread to add the details also to Host-pool service. If a new IP address is detected but with an existing hardware address, the same routine is called. The `register_a_new_host` function takes existing hosts into account and branches to patching function if need be. However, IP address changing for a host is a rare occasion and thus the function branching logic is done in Request Service for optimisation and maintaining source code readability.

Algorithm 4: get_id -function compares stored IP addresses and optionally hardware addresses to find out the corresponding Id number in Host-pool Service.

```
Input: ip_address
Input: hardware_address = None
stored_id = get_id(ip_address, hardware_address) if stored_id is
 None then
    data = *formatted json object corresponding to host's details*;
    response = *POST request to Host-pool Service with 'data' as
    payload.*;
    return response.json, message
end
else
    return RequestService.patch_a_host(stored_id, ip_address,
    hardware_address)
end
```

4.2.3 Limitations and assumptions of the Discovery Service

At its current prototypical state Discovery Service makes certain assumptions about the physical hosts in the cluster. Namely hosts are assumed to have Linux as an operating system (Distribution can vary) and a default user name and a password, those being 'centos' and 'admin' respectively. In addition hosts are required to be running an SSH server which is a requirement enforced to Linux hosts by Cloudify itself. Discovery Service requires that the standard port 22 for SSH is open.

4.2.4 Patching a host

Host-pool Service allows duplicate hosts, a feature which can be regarded as an oversight by the Host-pool Service developers. Discovery Service enforces that single host has only a single recorded IP address stored in the system. Therefore, if Discovery Service detects a host which has an already existing IP or Hardware Address, but the other one differs from the already stored one, Discovery Service will patch the given address with the new one. The patching to Redis is done in the Network Scanner as previously seen in algorithm 1. The patching to Host-pool Service is depicted in algorithm 5

Algorithm 5: patch_a_host function is called by register_a_new_host when Discovery Service detects a host whose IP address has changed or a new host which uses the same IP address as another host previously.

```
Input: id
Input: ip_address
Input: hardware_address
data = {};
host = RequestService.get_a_host(id);
if host.ip != ip_address then
    data['ip'] = ip_address;
end
else if host.name != hardware_address then
    data['name'] = hardware_address;
end
else
    return {}, message
end
response = *PATCH request to Host-pool Service with 'data' as
payload.*;
return response.json, message
```

The function `patch_a_host` takes three arguments: The Id of the host in Host-pool Service, host's IP address and its hardware. Then it retrieves the host's details from Host-pool Service in JSON format and compares the IP addresses received as an argument and retrieved from Host-pool Service. If they do not match, the argument IP address is patched as the new value to Host-pool Service using the REST function. In a rare case in which IP addresses match, but hardware addresses do not, the hardware address is patched to Host-pool Service as a name for the host. This is a highly unlikely event and in most cases happens because there are pre-configured hosts in the Host-pool Service and the database is not formatted before deploying Host-Pool Service causing the hosts to have names that do not conform to format enforced by Discovery Service i.e. names are hardware addresses.

4.3 Specification retriever

As seen on listing 1, Host-pool Service's data format has a list object named 'tags' which can hold an arbitrary number of arbitrary keywords used to describe a given host. Typical uses for these tags would be to describe a Linux distribution a host runs as the ’os’ object only accepts ’linux’ or ’windows’ or vaguely describe the physical capabilities of a host, for example ’small’ or ’large’. Host-pool plugin can use these tags to filter applicable hosts when they are allocated to Cloudify manager to use. The other attribute which can be used for filtering is the ’os’ key. The problem with the tags however is the fact that they like many other attributes in Host-pool service have to be applied manually.

As one of my goals is to allow Cloudify Manager to make more informed decisions based on hosts’ hardware capabilities and Host-pool plugin already having a filtering capabilities, it should be straightforward to extend that capability to also include hardware data. That however is not in the scope of this project. To leverage on hardware data, a way should exist to acquire it easily and automatically. That is why I have extended the Host-pool Service to contact the hosts added to the logical host-pool by the Discovery service and retrieve their individual hardware specifications. The source code for the custom Host-pool Service can be found here: `https://github.com/Fleuri/cloudify-host-pool-service`.

4.3.1 Technical implementation of the Specification Retriever

Specification retriever should retrieve actual hardware data from the target hosts, it should work without changing other functionality or user experience on Cloudify Manager, Host-pool Plugin, Host-pool Service or Discovery Service and it should work automatically without requiring user input or configuration.

These design goals in mind I have extended the Host-pool Service so that when a host is added to the logical host-pool, Host-pool Service runs a series of scripts on the given host which returns hardware data such as amount of RAM, number of CPUs and available disk space on that host. It also adds the received data to Host-pool Service’s data format so that it can be queried and patched using Host-pool Service’s REST API. The scripts can be run because each host’s data record includes their IP address, port and remote access protocol as well as access credentials. Due to prototypical limitations, the Specification Retriever presented in this thesis only works on Linux hosts as it uses Linux commands in the scripts.

```
    def run_command(self, command, spec_array, list_key, client):
        stdin, stdout, stderr = client.exec_command(command)
        line = stdout.readlines()
        spec_array[list_key] = line[0].strip('\n')

    def retrieve_hardware_specs(self, hosts):
        host = hosts[0]
        if host['os'] == 'linux':
            client = paramiko.SSHClient()
            client.set_missing_host_key_policy(paramiko.AutoAddPolicy())
            client.connect(host['endpoint']['ip'], port=host['endpoint']['port'],
                           username=host['credentials']['username'], password=host['credentials']['password'])
            spec_array = {}
            command_list = dict({'cpus': "lscpu | awk '/^CPU\(s\):/{ print $2}'",
                                 'ram': "free -m | awk '/Mem:/{ print $2}'",
                                 'disk_size': "df -h | awk '/\/$/{ print $2 }'",
                                 'graph_card_model': "lscpu | awk '/Model name/' | sed -e 's/Model name://g' -e 's/^[ \t]*//g'",
                                 'cpu_arch': "lscpu | awk '/Architecture/{ print $2 }'"
                                 })

        for list_key, command in command_list.items():
            self.run_command(command, spec_array, list_key, client)

        hosts[0]['hardware_specs'] = spec_array

```

```
442         client.close()
```

Listing 3: "Hardware specification retriever is an addition to Host-pool Service. Source code is more expressive than pseudocode in this particular case."

Specification Retriever is two additional functions implemented in Host-pool Service seen on listing 3. They are called the backend class' `add_host` seen on listing 4 which in turn is called whenever a new host is discovered in the network. Before Specification Retriever is run, the `HostAlchemist.parse()` function on line 283 checks the validity of the received host data and makes a couple of additions resulting in a data element similar to one depicted in listing 1. This element, *hosts* as seen on listing 5, is then passed to Specification retriever in which the hardware data is added to it.

```
277     def add_hosts(self, config):
278         '''Adds hosts to the host pool'''
279         self.logger.debug('backend.add_hosts({0})'.format(
                config))
280         if not isinstance(config, dict) or \
281            not config.get('hosts'):
282             raise exceptions.UnexpectedData('Empty hosts
                    object')
283         hosts = HostAlchemist(config).parse()
284         hosts = self.retrieve_hardware_specs(hosts)
285         return self.storage.add_hosts(hosts)
```

Listing 4: Host-pool Service's backend's add_host function makes a call to Specification Retriever

Specification Retriever's `retrieve_hardware_specs` function takes the *hosts* object as a parameter as it makes modifications to it and uses its data to perform the remote operations on host machines. After checking if the added host is a Linux host (In Discovery Service's case the OS is forced) the function establishes an SSH connection using the *Paramiko* library[21]. The arguments required to establish the connection, namely the host's IP address, port, user name and password are extracted from *hosts*. Next on line 430 there is a declaration for a key-value list consisting of keys that are to be added to *hosts* and matching Linux commands to extract the value for that key in the host system. Currently Specification Retriever uses the following commands:

lscpu to retrieve the number of CPU's in the host system.

free to acquire the amount of RAM the host system has.

df to list the overall disk space available in the system.

The Specification Retriever is designed so that adding new commands only requires listing them to `command_list` along with the identifying key.

After connecting to the host, the `run_command` routine on line 417 is ran for each command in the `command_list`. It runs the command on the host and reads the result from the host's standard output stream storing it into the `spec_array` hash table of results. After the for-loop `spec_array` is finally concatenated as a value for the key `hardware_specs` which is returned from the function after the connection is closed. The final list is added to the original data structure on line 284 seen in listing 4. The resulting data structure can be seen on listing 6 and it integrates seamlessly to existing data and functionality.

```
"hardware_specs": {
        "ram": "3219",
        "disk_size": "456G",
        "cpus": "2"
        }
```

Listing 5: An example of additional data inserted by the Specification Retriever

```
{
        "endpoint": {
                "ip": "192.168.150.2",
                "protocol": "ssh",
                "port": 22
                },
        "name": "08:9e:01:db:af:61",
        "tags": [],
        "alive": false,
        "hardware_specs": {
                "ram": "3372",
                "disk_size": "455G",
                "cpus": "2"
                },
        "credentials": {
                "username": "centos",
                "password": "admin"
                },
        "allocated": true,
        "os": "linux",
        "id": 1
}
```

Listing 6: An example of final data structure after the Specification Retriever has inserted hardware data

5 Experiments

During development, the Discovery Service and Specification Retriever software were tested separately by mocking other elements in the Cloudify cluster. This section describes the experiments and their set-ups used to verify that different components integrate and work together flawlessly in a full environment built on real machines. The components in question are the aforementioned Discovery Service and Specification Retriever in addition to Host-Pool Service, Cloudify Manager, Host-Pool Plugin and the test workload, Cloudify Nodecellar Example[6].

5.1 Hardware set-up

To verify that the Discovery Service and the Specification retriever function correctly in real environment and on real machines, I set up a test-bed depicted in figure 5.

The test-bed consist of a Lenovo Thinkpad T420S 4173L7G laptop computer with 4-core Intel i5-2540M CPU and 8 GB of RAM acting as a master node in the Cloudify cluster. The three slave-machines are Lenovo Thinkpad Edge E145's with 2-core AMD E1-2500 APU CPUs and 4 GB of RAM. The master host has Centos 7 installed as the operating system to accommodate Cloudify's installation requirements. The three slave machines are running Ubuntu 16.04 as the OS of the slaves can be anything as long as they are Linux-based and the hosts themselves can be accessed via SSH.

As seen on figure 5, the test-bed set up has two different networks. The Master can be accessed remotely via a bastion host *wint-13*. Wint-13 itself is not a part of the test-bed set up, but it is used to access the testbed remotely and allow internet access for and through the master host.

Master host has two network interfaces configured for each network it's a part of: The subnet A for outside access and B which is the subnet dedicated for the Cloudify cluster. Master also serves as a default gateway for all of the slaves. Figure 5 shows how slave machines are part of

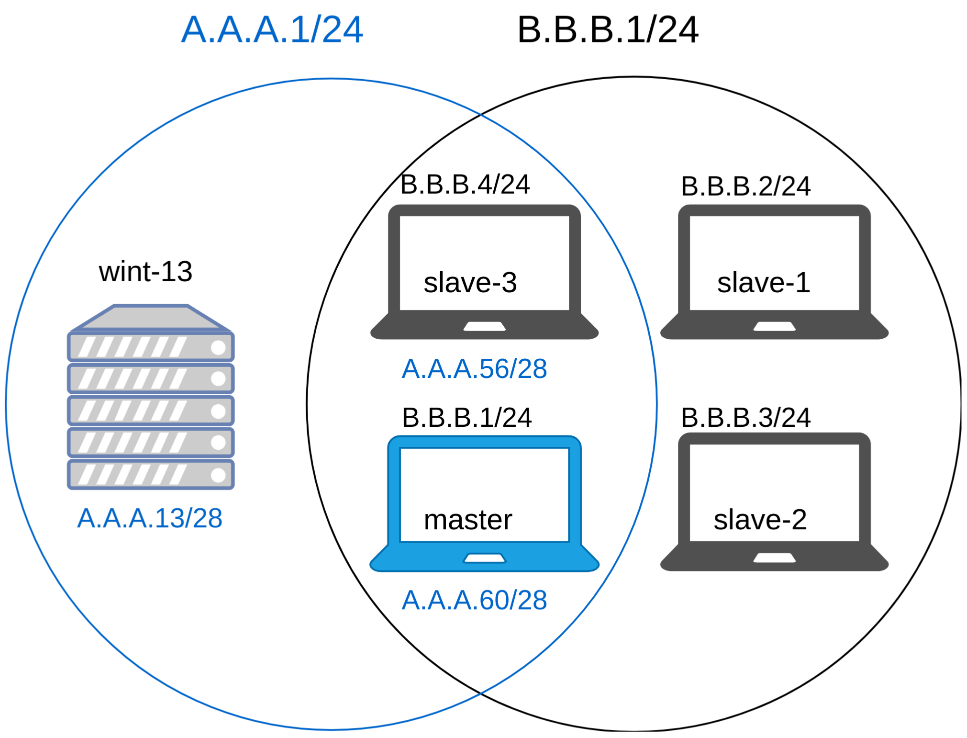


Figure 5: The testbed hosts are located in a private network but master and slave-3 are also directly connected to network A.

the cluster subnet with static IP addresses. In addition slave-3 is also connected to the A subnet. This is to provide a way to drop the host off and connect it back the cluster network but still be able to access it remotely via external network. The physical network is wired with Ethernet cables connected to an HP 5412 zl V2 switch.

5.2 Software environment set-up

As mentioned previously, slave hosts do not have many software requirements besides running a Linux distribution as an operating system and having an SSH server installed and accepting connections on the standard port 22. In the test-bed they also have their IP addresses and network interfaces configured statically.

Master node is more intricate than slaves in addition to being more powerful. It has the similar requirements to slaves when it comes to SSH server, but additionally it also runs the programs listed on table 1.

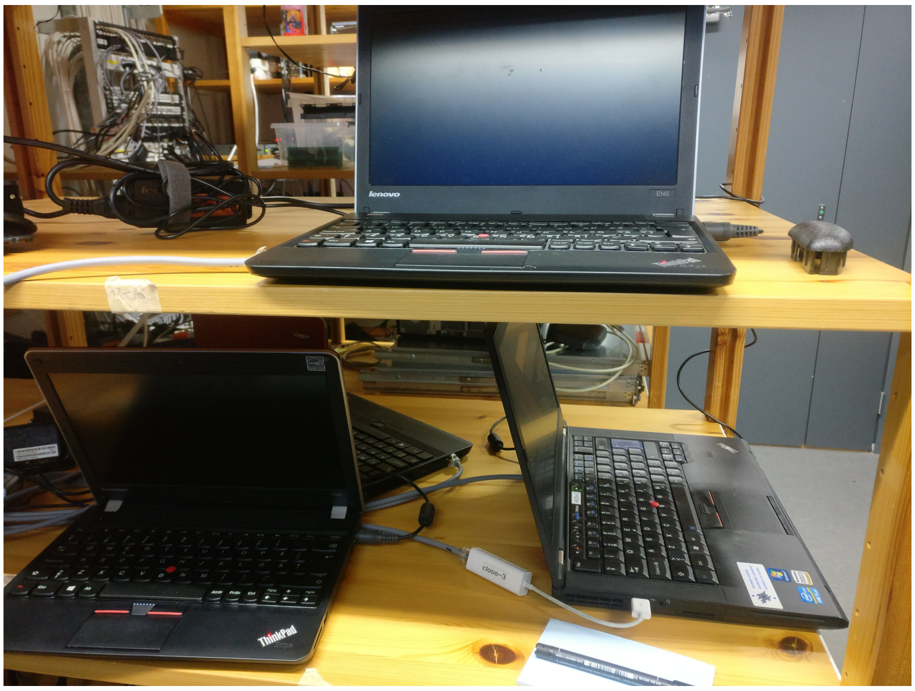


Figure 6: The actual test-bed. Master node on lower shelf on the right, slaves in the upper and lower shelf

Software	Version
Cloudify Manager	18.10.4 community
Host-pool Plugin within Cloudify Manager	1.4
Modified Host-pool Service	1.2
Discovery service	N/A
Docker	1.13.1
Redis running in a Docker container	5.0.0

Table 1: List of software and their versions used in the experiment setup

In addition the master host has to expose a certain number of ports both internally and externally listed on table 2.

Host-pool plugin runs as a Cloudify deployment managed by Cloudify manager. The modified Host-pool Service runs as a stand-alone Python program listening to port 5000. Even though Cloudify documentation recommends running Host-pool Service as a Cloudify Deployment on a separate host from Cloudify Manager, there are no drawbacks in this kind of local deployment either.

The Discovery Service is also run as standalone program and it doesn't reserve any ports. However the key-value storage the Discovery Service uses, Redis, is run in a docker container using the official Redis docker image. It reserves the standard Redis port 6379.

Finally, the server clocks on master host and slave-3 are synchronised with ntp against the ntp server located in network A, as synchronised time is needed for accurate measurement results in the experiments.

Application	Port
Cloudify ports	
Rest API and UI, HTTP	80
Rest API and UI, HTTPS	443
RabbitMQ	5671
Internal REST communication	53333
Other ports	
SSH	22
Host-pool Service*	5000
Redis*	6379
** Only internal access*	

Table 2: Required open ports on the master node. All ports are TCP

5.3 Test cases

To verify that different parts of the Discovery Service and Specification Retriever work in a real environment, I have come up with six test cases which test how different parts of the software integrate with a real system. First four of the test cases test Discovery Service's ability to monitor the cluster network and deliver the current cluster status to the Host-pool Service. The fifth test verifies that the Specification Retriever script in the Modified Host-pool service collects the hardware data correctly and also showcases its expandability. Finally I am going to run an example workload in the cluster which uses the Discovery Service to manage its logical host-pool. This verifies that the system can be used as an addition in a real Cloudify cluster.

The time measurements from all of the applicable test cases are displayed in table 3. The table shows the fastest and slowest measured times, average and median times as well as standard deviation. All timed tests are run thirty times and the detailed report of the measurement results can be found in appendix A.

5.3.1 Discovering hosts at start up

As detailed in the section 4.1.2, when Discovery Service is initialised it flushes all of the databases and performs an ARP scan for every IP address in the network. The time taking starts when the ARP scan itself starts and finishes after. The flushing of Host-pool Service's database and Redis are not included in the results. The scanning function imposes

Test-case	Min	Max	Mean	Median	Std. dev.
Start-up	5.40	5.79	5.59	5.58	0.094
Joining host	0.083	0.22	0.15	0.15	0.04
Parting host	40.39	45.14	42.51	42.47	1.34
Patching a host	0.04	0.06	0.048	0.048	0.004

Table 3: Summary of measurement results.

a two second wait time after the last packet is sent and the interval between each sent packet is 0,001 seconds.

Setting up the experiment

In this experiment I have modified the ARP scanning routine so that it measures the time it takes to scan through the 256 address network. The network itself contains three other hosts besides the master host which is ignored in the scan. This experiment does not require measurements from other servers and therefore no modifications or scripts are run on them. Additionally, the main function of Discovery Service is modified so that it runs the start-up routine thirty times and exits right after.

Results

As seen in the table 3, scanning through a 256 address network takes approximately 5.6 seconds. This means that it takes approximately 22 milliseconds to send and receive an ARP request for a single host. The real value however varies as the requests may return out of sync and with varying intervals. Also the `timeout` value which denotes the time spent waiting after the last request is set to relatively high value of two seconds.

Overall, all of the runs succeeded and there is no notable deviation in the distribution of times. In comparison to writing and providing host specification in JSON format, automatic scanning is significantly more efficient.

5.3.2 Detecting a joining host

One of the main features of the Discovery Service is the ability to detect machines joining the network in real time. In this test case Slave-3 is not initially connected to the cluster network. I have prepared a script which first returns a current time stamp on Slave-3 and then enables the network interface facing the cluster network. The sniffer algorithm on the Discovery Service is modified so that it too returns a time stamp upon detecting a new host. As both hosts are synced against the same time server, the timestamps are comparable allowing me to measure a time it takes for Discovery Service to detect a host after it has joined the network.

Setting up the experiment

This experiment required only a minor modification to Discovery Service's `register_a_new_host` function which printed the timestamp to a file whenever Slave-3 was detected as a new host. Slave-3 was initially disconnected from the cluster network. On slave-3 a BASH script was run which first turns on the network interface to network B, sleeps fifteen seconds and sends an ARP request to the network while recording the timestamp. Then the script sleeps for a minute, turns off the interface and waits another minute until and starts over. The function was repeated thirty times.

Results

As with the start-up scan, the time to detect a joining host is very regular and is more affected by the network speed rather than the implementation overhead. There were however outliers caused by the test implementation. As the network interface on Slave-3 was enabled there was a slight wait in the script execution so that the interface is ready before sending an ARP request. As the script was run multiple times, the ARP cache often did not have time to invalidate and thus no ARP request was automatically sent when the interface was ready, so manually sending the ARP request was necessary. In a real use case, such rapid enabling and disabling of the interface would be unlikely and the cache invalidation a non-issue. In few cases however, the cache was invalidated between a run and the ARP request was sent when interface was ready, causing the Discovery Service to catch the request before the time was recorded on Slave-3 resulting in a negative time in the final results. Those times have been disregarded in the table 3 but are provided in the appendix A.

Overall most of time taken to detect a joining hosts consists of the interval between sending and receiving the ARP request.

5.3.3 Detecting a departed host

Similarly to detecting the joining host, detection of a departed host is another major feature of the Discovery Service. The testing procedure is also similar: Slave-3 has a script which drops the host from the network while producing a time stamp for the event. The departure detection in the Discovery Service is modified to return a time stamp when the detection of Slave-3 is detected. The detection is done in the pinger component described in section 4.1.3, in which a host is declared departed if it fails to reply to a set number of pings. The values for the ping interval and ping failures are five seconds for the interval and three failures. This relatively long interval is likely to cause a wide distribution of time results as the time between Slave-3 getting dropped from the network and the first ping could be five seconds. On the other hand this measurement is representable of a real usage scenario.

Setting up the experiment

As in the previous experiment, only modification done to Discovery Service is producing a timestamp when Slave-3 is declared to have left the network. The script run on Slave-3 is also virtually identical to that of experiment 5.3.2 with the exception that the time stamp is recorded when the network interface is disabled. Slave-3 started disconnected from the cluster network in this case also.

Results

Every execution of the pinging routine consists of two parts which make up the majority of the execution time. First the time out the ARP ping spends waiting for a reply is ten seconds and an interval between pings is five seconds. Depending on how fast a ping routine fires after Slave-3 disconnects, these parts take minimum of forty seconds and maximum of 45 five seconds.

Both extremums are presented in the experiment sampling and both average and median values are close to expected mathematical average. The computational overhead is negligible when compared to the ping interval and time out, but keeping that in mind, the average and median times could have been expected to be slightly over 42,5 seconds. Nevertheless, taking into account the expected five second variability of the beginning of execution, another data sample could result in slightly different times but similar standard deviation.

5.3.4 Patching a host

Procedure on this test case is similar to the previous two. A script modifies the IP address of Slave-3 from B.B.B.4 to B.B.B.5 and back, each modification producing a time stamp for in Slave-3 and each detected address change producing one on Master. Similarly to *Detecting a joining host* experiment, an ARP request is sent manually as the ARP caches may not have time to be invalidated and thus changing the IP address may not trigger an ARP request automatically.

Setting up the experiment

The experiment setup started similarly to previous two but instead of one function on Slave-3 to turn the network interface on and off, there needed to be two to alternate between .4 and .5 IP addresses using the network manager. Discovery Service's `patch_a_host` function was modified to produce a timestamp whenever Slave-3's IP address was edited.

During the preliminary testing it became apparent that in some cases, as one IP address was unused for a longer time than for example in *Detecting a parting host* experiment, ARP caches became invalidated more frequently. This meant, that changing an IP address triggered an automatic ARP call more frequently than first expected and capturing this behaviour required an another time stamp to be recorded in the script on Slave-3.

Results

In the experiment, Slave-3 provided time stamp both when the interface was enabled and also when the ARP request was sent manually. The first time stamp prevented negative times in the results but the time itself didn't reflect a time it takes for the Discovery Service to detect and patch an IP change, but rather the time it takes for a the interface to be ready in this particular case. These values have been omitted from the results in table 3 but they are included in the appendix A.

The tests succeeded on every execution. Patching seemed to be the most lightweight of the operations tested: The slowest execution time was circa 26 milliseconds faster than the fastest execution of the *Detecting a joining host* experiment. This is because changing a single field in an existing data object is computationally significantly less demanding than creating a new data object with multiple fields.

Like the results from the previous experiments, *Patching a host* results also indicate correct, fast and regular execution of the routine.

5.3.5 Retrieving hardware data from the hosts

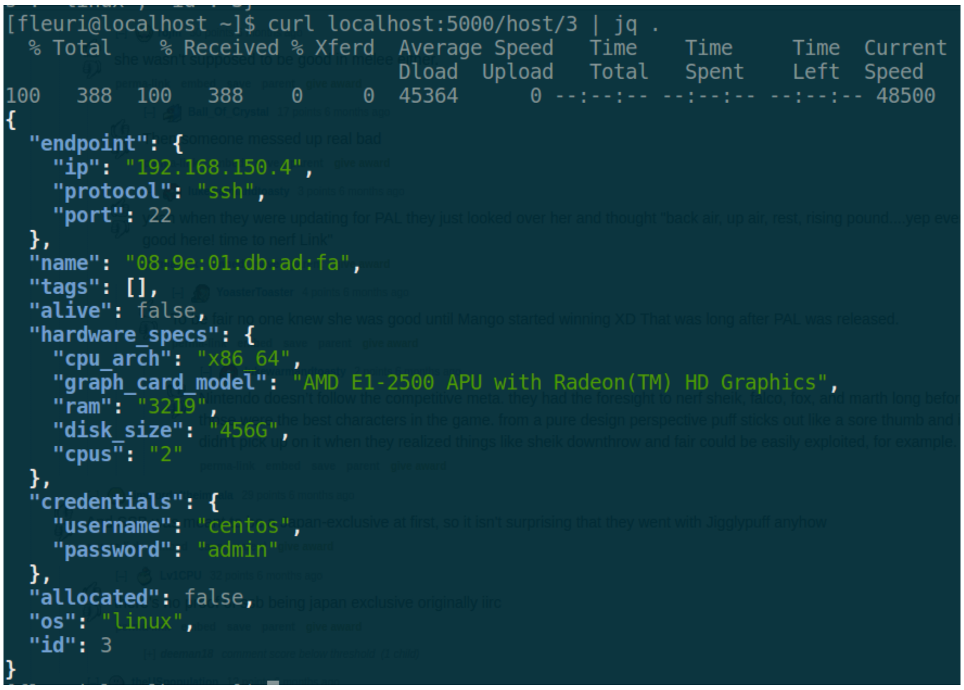
This test case shows, that the modified Host-pool Service retrieves correct hardware data from the host. First the hardware data is listed manually on the target host. Next the start-up routine is run which adds all the hosts in the network to logical host pool. The modified Host-pool Service runs the hardware specification retrieval scripts when the hosts are added. Afterwards Host-pool Service can be queried to confirm that correct data was retrieved.

This test case also demonstrates how easily new commands can be added to the Specification Retriever.

Setting up the experiment

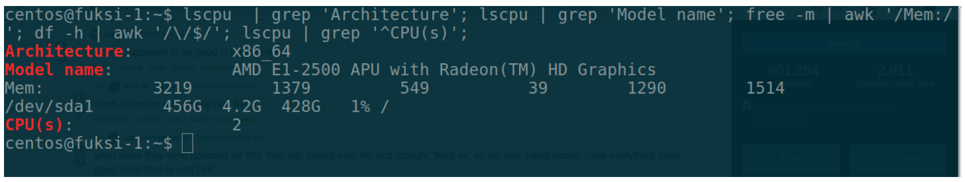
This experiment did not require any additional modifications apart from additional commands described in the next section. However, correct values from Slave-3 were retrieved by hand as seen on figure 8 so that the values produced by the Specification Retriever could be verified. As values were static and slave hosts identical, the experiment was run only once.

Results



```
[fleuri@localhost ~]$ curl localhost:5000/host/3 | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   388  100   388    0     0  45364      0 --:--:-- --:--:-- --:--:-- 48500
{
  "endpoint": {
    "ip": "192.168.150.4",
    "protocol": "ssh",
    "port": 22
  },
  "name": "08:9e:01:db:ad:fa",
  "tags": [],
  "alive": false,
  "hardware_specs": {
    "cpu_arch": "x86_64",
    "graph_card_model": "AMD E1-2500 APU with Radeon(TM) HD Graphics",
    "ram": "3219",
    "disk_size": "456G",
    "cpus": "2"
  },
  "credentials": {
    "username": "centos",
    "password": "admin"
  },
  "allocated": false,
  "os": "linux",
  "id": 3
}
```

Figure 7: Host-pool service query returns a json with added new fields cpu_arch and graph_card_model



```
centos@fuksi-1:~$ lscpu  | grep 'Architecture'; lscpu | grep 'Model name'; free -m | awk '/Mem:/
'; df -h | awk '/\/$/'; lscpu | grep '^CPU(s)';
Architecture:          x86_64
Model name:            AMD E1-2500 APU with Radeon(TM) HD Graphics
Mem:           3219        1379         549          39        1290        1514
/dev/sda1        456G  4.2G  428G   1% /
CPU(s):                2
centos@fuksi-1:~$ 
```

Figure 8: The results of individual commands run on Slave-3

OpenStack's *Ironic* -project[28] retrieves a modest amount of hardware data, namely the Number of CPUs, available RAM, available disk space and CPU architecture. CPU architecture retrieval was not included in the modified Host-pool Service specification detailed in section 4.3, but to demonstrate specification retrievers capabilities and extendibility, I have added it as well as a command to retrieve the host's graphics card model to the list of commands executed on hosts. The commands are as follows:

- `lscpu | awk '/Architecture/{ print \$2 }'` which retrieves the cpu architecture.
- `lscpu | awk '/Model name/' |`
  `sed -e 's/Model name://g' -e 's/^[ \t]*//g'` which retrieves the graphics card model while removing whitespace from the command result.

In the experiment the additional query commands were added to the modified Host-pool service. Discovery Service was ran normally and the Host-Pool Service REST endpoint was queried for Slave-3's data. Figure 7 is a screenshot of the query result and it shows the hardware data along with the added CPU architecture and graphics card model. Figure 8 is a screenshot showing the same data when queried directly on Slave-3 in remote terminal. The data in Host-pool service query is identical to that queried directly on Slave-3, verifying the correctness of the Hardware Specification retrieval operation. It also verifies, that the Hardware Specification retrieval operation can be easily extended with only minor additions to Host-pool Service.

5.3.6 Running an example workload in the system

The final experiment shows, that Discovery Service works seamlessly when running a real workload in the Cloudify cluster. The workload in question is Cloudify's standard example workload Cloudify Nodecellar Example[6]. Nodecellar is a web application which simulates a wine inventory system. It is deployed on two separate hosts with the *nodejs_host*

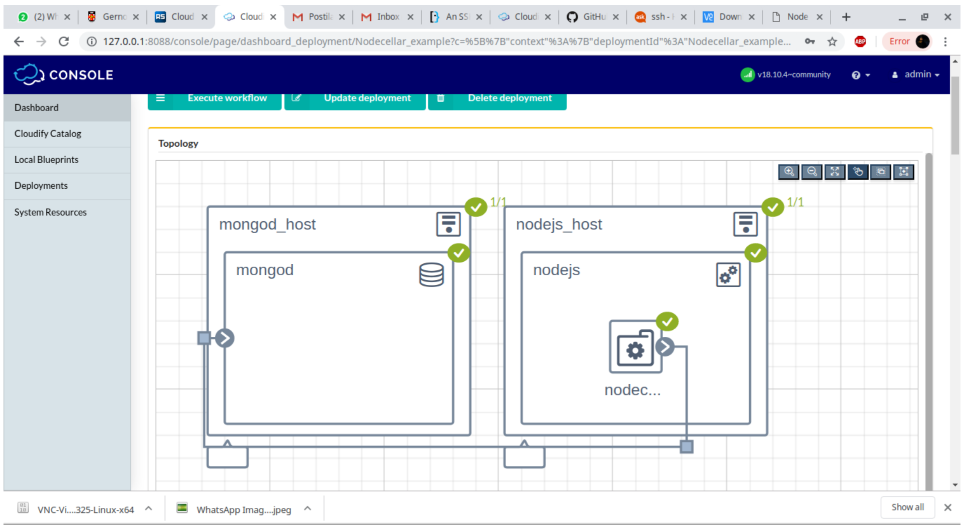
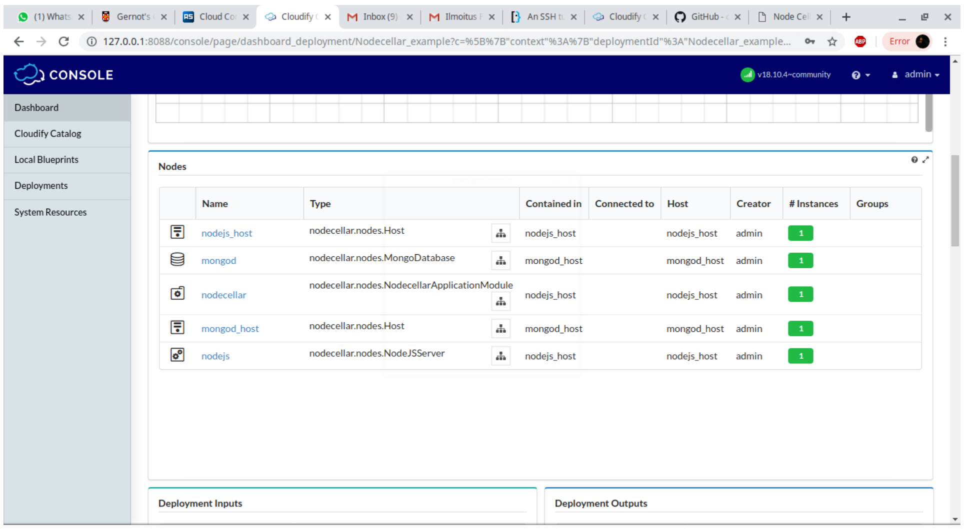


Figure 9: Cloudify console shows the topology diagram of the Nodecellar deployment



Nodes

	Name	Type	Contained in	Connected to	Host	Creator	# Instances	Groups
	nodejs_host	nodecellar.nodes.Host	nodejs_host		nodejs_host	admin	1	
	mongod	nodecellar.nodes.MongoDatabase	mongod_host		mongod_host	admin	1	
	nodecellar	nodecellar.nodes.NodecellarApplicationModule	nodejs_host		nodejs_host	admin	1	
	mongod_host	nodecellar.nodes.Host	mongod_host		mongod_host	admin	1	
	nodejs	nodecellar.nodes.NodeJSServer	nodejs_host		nodejs_host	admin	1	

Figure 10: Nodecellar deployment’s parts listed in the Cloudify console

running a webserver and a Node.js based frontend application whereas the second *mongod_host* houses MongoDB. The figures 9 and 10 are screenshots of the Cloudify console showing the different components of the Nodecellar deployment and the relations between them. The goal of this experiment is to have a functioning Nodecellar application running on two of the slave hosts which were discovered by the Discovery Service and allocated to Cloudify Manager by the Host-Pool Service.

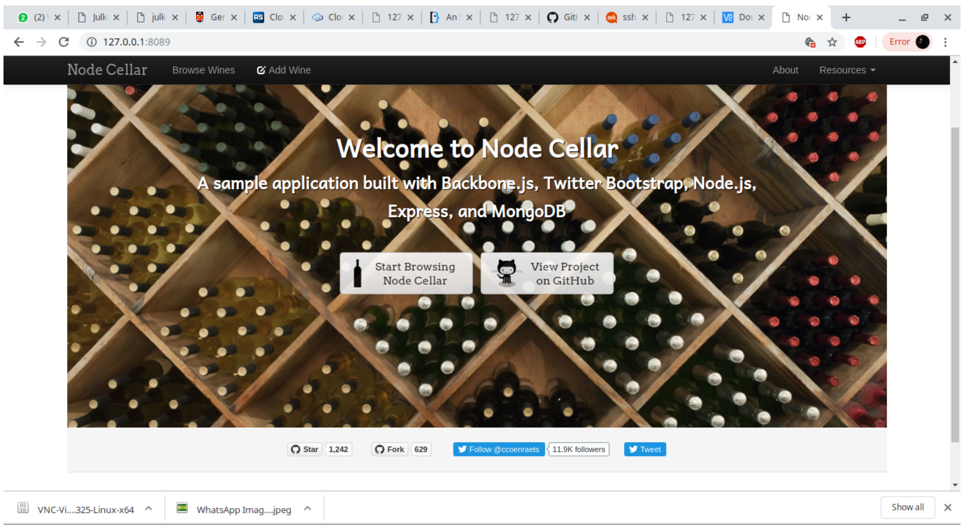


Figure 11: Cloudify Nodecellar Example's landing page when deployed on the test bed cluster. The localhost IP address is due to port being forwarded to the *node_js* host.

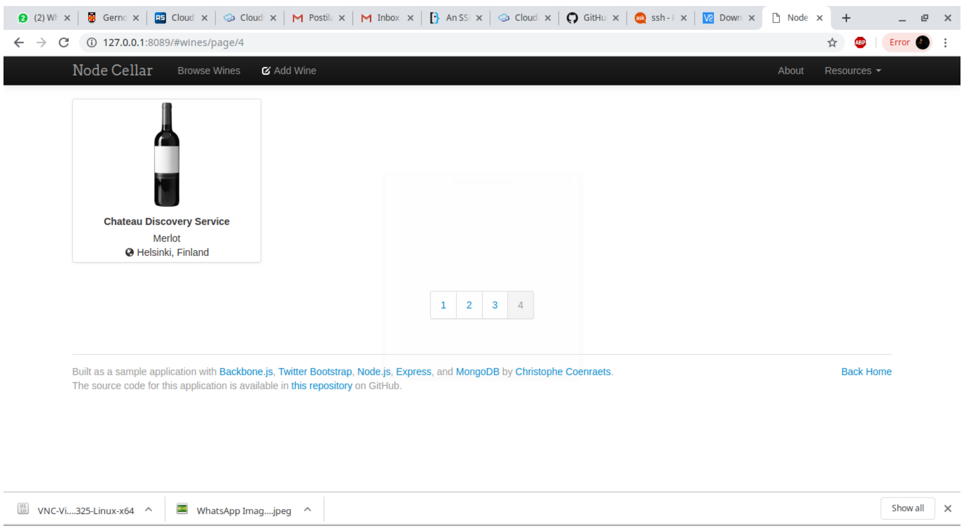

127.0.0.1:8089/#wines/page/4
Node Cellar
Browse Wines
Add Wine
About
Resources
Chateau Discovery Service
Merlot
Helsinki, Finland
1 2 3 4
Built as a sample application with Backbone.js, Twitter Bootstrap, Node.js, Express, and MongoDB by Christophe Coenraets.
The source code for this application is available in this repository on GitHub.
Back Home
VNC-Vi....325-Linux-x64
WhatsApp Imag....jpeg
Show all

Figure 12: Nodecellar allows the user to add their own wines to the list. This particular vintage is a real labour of love.

Prior to deploying Nodecellar, I had set up Host-pool Service, Discovery Service and Cloudify Manager. I had also forwarded the port of Cloudify Manager's web console so that I could access it remotely. On

Cloudify, every component has to be defined as a Blueprint using TOSCA DSL. When blueprints are uploaded to Cloudify Manager, user can create Deployments of them which in turn create the actual resources.

Before Cloudify Manager could use Host-pool Service to allocate hosts for the deployment, the Host-pool Plugin was required to be installed. The latest version 1.5 was faulty so I downgraded to 1.4. This also meant that I had to manually change Nodecellar Example's blueprint's dependencies to use Host-pool Plugin version 1.4 instead of the default 1.5. After uploading the modified blueprints and creating deployments of them I ran the 'Install' workflow which requested the required hosts from Host-pool Service via Host-pool Plugin and after receiving them installed the required software on them.

Results

As Host-pool Plugin allocates the requested available hosts in order, Slave-1 was designated as the *nodejs_host* and Slave-2 as *mongod_host*. To verify that the application is usable and served correctly I forwarded another port via wint-13 bastion host and Master host so that I could view the web page on my local machine. Figure 11 shows a screenshot of the Nodecellar application. I could also add and remove wines to and from the system as seen on figure 12. This validated that the MongoDB database on the second host as well as the connection between the hosts functioned correctly.

Overall it can be concluded that Discovery Service and the modified Host-pool Service function correctly in the experimental scope of this thesis.

6 Future Research and Conclusions

As seen in the previous section, Discovery Service and Modified Host-pool Service work well within their their limited prototypical scope. There are however both major and minor assumptions, lacking features and programming solutions that should be addressed before the software could be regarded as a mature, full-fledged solution.

As of now, Discovery Service does the bare minimum of error checking and, while no errors or bugs were detected while evaluating the system, there are some known possibilities for failure e.g. network communication failure between Discovery Service and Host-pool service would prevent Host-pool service from updating. There's also no robust logging in the system. Maybe the most major design issue in the system is the two different data formats with Redis having their own data for hosts and Host-pool service their own. This decision was made early in the project and its implications became apparent only later on. Granted, Discovery Service does not need as much data on the hosts than the Host-pool Service, but redesigning its data format to resemble that of Host-pool Service's could bring clarity to the source code and eliminate the need to match different primary keys (MAC addresses in Discovery Service, running count in Host-pool Service). The major assumptions of Discovery Service are that hosts are Linux hosts and that they have a common username and password. To mature Discovery Service, it should also work with Windows hosts. Key and credentials handling is a larger and more complex challenge and would likely require preparing the hosts somehow, like installing software on them beforehand.

The Modified Host-pool Service's Specification Retriever also relies on assumption that the hosts in the network run Linux, but it does check for the fact before trying to run scripts on them. However it also makes an assumption that the commands it runs on the hosts exist on them. A production ready version should check if the command can be run before attempting to run it. It should also be able to support Windows hosts.

An issue I noticed while working with Host-pool Service is that the *alive* field is never used and by default all hosts are marked dead. Discovery Service could, instead of immediately removing an entry when a host leaves the network, manipulate the alive field.

To be more in the line with other Cloudify components, Discovery Service should be packaged as a Cloudify Blueprint.

Finally, a larger scale performance testing could uncover issues that are not apparent in the prototypical scope of the works. Another project could be work on the Cloudify Manager to leverage on more detailed hardware data Host-pool Service now provides so that it could make more intelligent scheduling and provisioning decisions.

Otherwise I have created groundwork for automating and managing a Cloudify cluster consisting of generic and heterogeneous hosts and provided a mechanism to gather performance and hardware related data from them. This allows users to easily introduce varied infrastructure to their cloud computing cluster and account for hardware differences between the hosts while eliminating some aspects of manual work in the cluster management and enabling 'plug-and-play' approach to adding and removing hosts. The possibility to use heterogeneous hardware allows user to size their hardware capabilities, heat and energy efficiency and costs to fit their workloads with Cloudify enabling operation in hybrid environments.

Sources

[1] *Canonical maas*. https://maas.io/. Accessed 19.5.2019.

[2] *Cloud foundry*. https://www.cloudfoundry.org. Accessed 25.5.2019.

[3] *Cloudify*. https://cloudify.co/. Accessed: 31.5.2019.

[4] *Cloudify faq*. https://cloudify.co/FAQ_cloud_devops_automation#q05A. Accessed 25.5.2019.

[5] *Cloudify host-pool service*. https://github.com/cloudify-cosmo/cloudify-host-pool-service. Accessed: 31.5.2019.

[6] *Cloudify nodecellar example*. https://github.com/cloudify-cosmo/cloudify-nodecellar-example. Accessed 27.1.2019.

[7] *Companies supporting the openstack foundational*. https://www.openstack.org/foundation/companies/. Accessed: 31.5.2019.

[8] *Docker*. https://www.docker.com. Accessed: 31.5.2019.

[9] *Docker machine*. https://docs.docker.com/machine/. Accessed 25.5.2019.

[10] *Heat*. https://docs.openstack.org/heat/latest/. Accessed 26.1.2019.

[11] *Ibm hybrid, multicloud management*. https://www.ibm.com/cloud/management. Accessed 5.5.2019.

[12] *ironic*. https://wiki.openstack.org/wiki/Ironic. Accessed 19.5.2019.

[13] *jclouds*. https://jclouds.apache.org/. Accessed 25.5.2019.

[14] *Kubernetes*. https://kubernetes.io/. Accessed 26.1.2019.

[15] *Microsoft Hyper-V*. https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2012-R2-and-2012/mt169373(v=ws.11). Accessed: 31.5.2019.

[16] *MirageOS*. https://mirage.io/. Accessed: 31.5.2019.

[17] *Openstack*. https://www.openstack.org/. Accessed: 31.5.2019.

[18] *Openstack: Features and benefits*. https://docs.openstack.org/swift/stein/admin/. Accessed: 31.5.2019.

[19] *Oracle VirtualBox*. https://www.virtualbox.org/. Accessed: 31.5.2019.

[20] *Packet*. https://packet.com/. Accessed 5.5.2019.

[21] *Paramiko*. http://www.paramiko.org/. Accessed 12.12.2018.

[22] *Rancher*. https://www.rancher.com. Accessed 5.5.2019.

[23] *Raspberry pi*. https://www.raspberrypi.org/. Accessed: 31.5.2019.

[24] *Razor*. https://puppet.com/docs/pe/2017.1/razor_intro.html. Accessed 19.5.2019.

[25] *Red hat satellite documentation: Provisioning guide chapter 5: Provisioning bare metal hosts*. https://access.redhat.com/documentation/en-us/red_hat_satellite/6.4/html/provisioning_guide/provisioning_bare_metal_hosts. Accessed 25.5.2019.

[26] *Redis*. https://redis.io/. Accessed 31.5.2019.

[27] *Scapy*. https://scapy.net/. Accessed 27.10.2018.

[28] *Troubleshooting ironic*. https://docs.openstack.org/ironic/pike/admin/troubleshooting.html#top. Accessed 7.4.2019.

[29] *VMware VSphere Hypervisor*. http://www.vmware.com/products/vsphere-hypervisor.html. Accessed: 31.5.2019.

[30] *VMware workstation*. http://www.vmware.com/products/workstation-pro.html. Accessed: 31.5.2019.

[31] *Vultr*. https://vultr.com/. Accessed 5.5.2019.

[32] *Xen project*. https://www.xenproject.org/. Accessed: 31.5.2019.

[33] *Tosca simple profile in yaml version 1.1*. Oasis standard, January 2018. Latest version: http://docs.oasis-open.org/tosca/TOSCA-Simple-Profile-YAML/v1.1/TOSCA-Simple-Profile-YAML-v1.1.html.

[34] Arpaci-Dusseau, Remzi H. and Arpaci-Dusseau, Andrea C.: *Operating Systems: Three Easy Pieces*. Arpaci-Dusseau Books, 0.91 edition, May 2015.

[35] Cheshire, Dr. Stuart D.: *IPv4 Address Conflict Detection*. RFC 5227, July 2008. `https://rfc-editor.org/rfc/rfc5227.txt`.

[36] Clark, Christopher, Fraser, Keir, H, Steven, Hansen, Jacob Gorm, Jul, Eric, Limpach, Christian, Pratt, Ian, and Warfield, Andrew: *Live migration of virtual machines*. In *In Proceedings of the 2nd ACM/USENIX Symposium on Networked Systems Design and Implementation (NSDI*, pages 273–286, 2005.

[37] Crago, S., Dunn, K., Eads, P., Hochstein, L., Kang, D. I., Kang, M., Modium, D., Singh, K., Suh, J., and Walters, J. P.: *Heterogeneous cloud computing*. In *2011 IEEE International Conference on Cluster Computing*, pages 378–385, Sept 2011.

[38] Duan, Y., Fu, G., Zhou, N., Sun, X., Narendra, N. C., and Hu, B.: *Everything as a service (xaas) on the cloud: Origins, current and future trends*. In *2015 IEEE 8th International Conference on Cloud Computing*, pages 621–628, June 2015.

[39] Eder, Michael: *Hypervisor-vs. container-based virtualization*. Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM), 1, 2016.

[40] Ferry, N., Rossini, A., Chauvel, F., Morin, B., and Solberg, A.: *Towards model-driven provisioning, deployment, monitoring, and adaptation of multi-cloud systems*. In *2013 IEEE Sixth International Conference on Cloud Computing*, pages 887–894, June 2013.

[41] Flexera: *Rightscale 2019 State of The Cloud Report from Flexera*. Technical report, February 2019.

[42] Jadeja, Y. and Modi, K.: *Cloud computing - concepts, architecture and challenges*. In *2012 International Conference on Computing, Electronics and Electrical Technologies (ICCEET)*, pages 877–880, March 2012.

[43] Jimmy McArthur, Alison Price et al.: *2018 openstack user survey report*. Technical report, 2018.

[44] Kivity, Avi, Laor, Dor, Costa, Glauber, Enberg, Pekka, Har'El, Nadav, Marti, Don, and Zolotarov, Vlad: *OSv—optimizing the operating system for virtual machines*. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*, pages 61–72, Philadelphia, PA, June 2014. USENIX Association, ISBN 978-1-931971-10-2. `https://www.usenix.org/conference/atc14/technical-sessions/presentation/kivity`.

[45] Kurp, Patrick: *Green computing*. Commun. ACM, 51(10):11–13, October 2008, ISSN 0001-0782. `http://doi.acm.org/10.1145/1400181.1400186`.

[46] Luszczek, Piotr, Meek, Eric, Moore, Shirley, Terpstra, Dan, Weaver, Vincent M., and Dongarra, Jack: *Evaluation of the hpc challenge benchmarks in virtualized environments*. In Alexander, Michael, D'Ambra, Pasqua, Belloum, Adam, Bosilca, George, Cannataro, Mario, Danelutto, Marco, Di Martino, Beniamino, Gerndt, Michael, Jeannot, Emmanuel, Namyst, Raymond, Roman, Jean, Scott, Stephen L., Traff, Jesper Larsson, Vallée, Geoffroy, and Weidendorfer, Josef (editors): *Euro-Par 2011: Parallel Processing Workshops*, pages 436–445, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg, ISBN 978-3-642-29740-3.

[47] Madhavapeddy, Anil, Mortier, Richard, Rotsos, Charalampos, Scott, David, Singh, Balraj, Gazagnaire, Thomas, Smith, Steven, Hand, Steven, and Crowcroft, Jon: *Unikernels: Library operating systems for the cloud*. SIGPLAN Not., 48(4):461–472, March 2013, ISSN 0362-1340. `http://doi.acm.org/10.1145/2499368.2451167`.

[48] Mell, Peter M. and Grance, Timothy: *Sp 800-145. the nist definition of cloud computing*. Technical report, Gaithersburg, MD, United States, 2011.

[49] Porter, Donald E., Hunt, Galen, Howell, Jon, Olinsky, Reuben, and Boyd-Wickizer, Silas: *Rethinking the library os from the top down*. In *Proceedings of the 16th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. Association for Computing Machinery, Inc., March 2011. `https://www.microsoft.com/en-us/research/publication/rethinking-the-library-os-from-the-top-down/`.

[50] Rad, P., Chronopoulos, A. T., Lama, P., Madduri, P., and Loader, C.: *Benchmarking bare metal cloud servers for hpc applications*. In *2015 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM)*, pages 153–159, Nov 2015.

[51] Sahoo, J., Mohapatra, S., and Lath, R.: *Virtualization: A survey on concepts, taxonomy and associated security issues*. In *Computer and Network Technology (ICCNT), 2010 Second International Conference on*, pages 222–226, April 2010.

[52] Sefraoui, Omar, Aissaoui, Mohammed, Eleuldj, Mohsine, Iaas, Openstack, and Scalableifx, Virtualization: *Openstack: Toward an open-source solution for cloud computing*, 2012.

[53] Shan, Yizhou, Huang, Yutong, Chen, Yilun, and Zhang, Yiying: *Legoos: A disseminated, distributed OS for hardware resource disaggregation*. In *2019 USENIX Annual Technical Conference (USENIX ATC 19)*, pages 69–87, Renton, WA, 2019. USENIX Association, ISBN 978-1-931971-47-8. `https://www.usenix.org/conference/atc19/presentation/shan`.

[54] Xavier, M. G., Neves, M. V., Rossi, F. D., Ferreto, T. C., Lange, T., and Rose, C. A. F. De: *Performance evaluation of container-based virtualization for high performance computing environments*. In *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 233–240, Feb 2013.

A Test measurements

A.1 All start-up scan times

Run order	Time taken in seconds		
1	5.5212135315	Average:	5.5874319077
2	5.6992228031	Median:	5.5766154528
3	5.7459275723	Fastest:	5.402588129
4	5.5203478336	Slowest:	5.7881305218
5	5.402588129	Standard Deviation::	0.0941250689
6	5.5956482887		
7	5.5960564613		
8	5.6395816803		
9	5.5402934551		
10	5.5746655464		
11	5.5216667652		
12	5.5795629025		
13	5.7881305218		
14	5.6899979115		
15	5.5003581047		
16	5.6109347343		
17	5.5516016483		
18	5.6865930557		
19	5.7207539082		
20	5.6052863598		
21	5.5212519169		
22	5.7254798412		
23	5.6741354465		
24	5.5785653591		
25	5.5100474358		
26	5.5130033493		
27	5.4387583733		
28	5.5147643089		
29	5.5006010532		
30	5.555918932		

A.2 All host discovery times

Run order	Host joined network	Host found	Time taken in seconds
1	1551985222.15239	1551985222.37099	0.2186000347
2	1551985353.37076	1551985353.55532	0.1845600605
3	1551985484.60706	1551985484.69028	0.083220005
4	1551985615.83384	1551985615.97919	0.1453502178
5	1551985747.0425	1551985747.13216	0.0896599293
6	1551985878.28073	1551985868.89678	-9.383949995
7	1551986009.47946	1551986009.63293	0.1534700394
8	1551986140.69389	1551986140.78502	0.0911300182
9	1551986271.93329	1551986272.12196	0.18866992
10	1551986403.17087	1551986393.87112	-9.2997500896
11	1551986534.3891	1551986525.00583	-9.3832700253
12	1551986665.60635	1551986665.80476	0.1984100342
13	1551986796.82454	1551986796.97268	0.148140192
14	1551986928.07304	1551986928.22165	0.1486098766
15	1551987059.31184	1551987049.88948	-9.4223599434
16	1551987190.55872	1551987190.74462	0.1858999729
17	1551987321.77601	1551987321.88523	0.1092200279
18	1551987453.02166	1551987453.16549	0.1438298225
19	1551987584.23665	1551987574.87545	-9.3612000942
20	1551987715.47122	1551987715.67965	0.2084300518
21	1551987846.72406	1551987846.87137	0.1473100185
22	1551987977.94288	1551987978.13356	0.190680027

With negatives times	
Average:	-2.0597443342
Median:	0.1445900202
Fastest:	-9.4223599434
Slowest:	0.2186000347
Standard Deviation:	4.0779027162
Without negatives times	
Average:	0.1519426159
Median:	0.1486098766
Fastest:	0.083220005
Slowest:	0.2186000347
Standard Deviation:	0.0406485995

23	1551988109.16361	1551988099.90049	-9.263119936
24	1551988240.37602	1551988240.55236	0.1763401031
25	1551988371.62402	1551988371.75052	0.1264998913
26	1551988502.85096	1551988503.004	0.1530399323
27	1551988634.07713	1551988624.90377	-9.1733601093
28	1551988765.32055	1551988765.51199	0.1914401054
29	1551988896.54041	1551988896.63656	0.0961499214
30	1551989027.77744	1551989027.89346	0.1160199642

A.3 All host disconnection times

Run Order	Host Disconnected	Host Deregistered	Time taken in seconds
1	1552122838.30771	1552122878.84452	40.5368101597
2	1552122969.52821	1552123010.39057	40.8623600006
3	1552123100.75055	1552123141.92967	41.1791200638
4	1552123232.00522	1552123273.43764	41.4324200153
5	1552123363.2187	1552123404.94401	41.7253100872
6	1552123494.41722	1552123536.44338	42.0261600018
7	1552123625.64917	1552123667.99715	42.3479800224
8	1552123756.89979	1552123799.49884	42.599050045
9	1552123888.12656	1552123930.99032	42.8637599945
10	1552124019.33029	1552124062.49085	43.1605598927
11	1552124150.55396	1552124193.94822	43.3942599297
12	1552124281.7748	1552124325.49912	43.7243199348
13	1552124413.01101	1552124457.04074	44.0297300816
14	1552124544.22855	1552124588.53997	44.3114199638
15	1552124675.45835	1552124720.04886	44.5905101299
16	1552124806.6943	1552124851.50203	44.8077299595
17	1552124937.91599	1552124983.05333	45.1373398304
18	1552125069.14882	1552125109.54138	40.3925600052
19	1552125200.36688	1552125241.03826	40.671380043
20	1552125331.59595	1552125372.52238	40.9264302254
21	1552125462.8262	1552125504.02815	41.2019500732
22	1552125594.05397	1552125635.49851	41.4445397854

Average:	42.5071433465
Median:	42.4735150337
Fastest:	40.3925600052
Slowest:	45.1373398304
Standard Deviation:	1.3414346833

23	1552125725.2785	1552125767.03705	41.7585499287
24	1552125856.50981	1552125898.53611	42.0262999535
25	1552125987.75468	1552126030.04648	42.2918000221
26	1552126118.97288	1552126161.59271	42.6198301315
27	1552126250.22228	1552126293.09072	42.8684399128
28	1552126381.4452	1552126424.58656	43.1413600445
29	1552126512.69665	1552126556.13836	43.4417099953
30	1552126643.93728	1552126687.63789	43.7006101608

A.4 All host patching times

Run Order	Host patched	Arp sent after interface ready	Arp sent manually	Time taken in seconds
1	1554090361.33807	1554090351.28472	1554090361.29293	0.045140028
2	1554090436.94233	1554090426.88258	1554090436.8893	0.0530297756
3	1554090512.57653	1554090502.51601	1554090512.53004	0.046489954
4	1554090588.14319	1554090578.07924	1554090588.09521	0.0479798317
5	1554090663.74458	1554090653.68545	1554090663.69425	0.0503299236
6	1554090734.77337	1554090729.30577	1554090739.31745	5.4676001072
7	1554090814.9184	1554090804.86416	1554090814.87155	0.046849966
8	1554090890.50323	1554090880.44033	1554090890.44953	0.0537002087
9	1554090966.06846	1554090956.01977	1554090966.02604	0.0424199104
10	1554091041.64256	1554091031.57167	1554091041.58865	0.053910017
11	1554091117.22359	1554091107.16494	1554091117.178	0.0455899239
12	1554091192.83752	1554091182.76329	1554091192.78027	0.0572497845
13	1554091268.44217	1554091258.37663	1554091268.39109	0.0510799885
14	1554091344.02731	1554091333.96437	1554091343.97667	0.0506398678
15	1554091419.64087	1554091409.58761	1554091419.59896	0.0419101715
16	1554091490.80174	1554091485.17243	1554091495.18706	5.6293098927
17	1554091570.82652	1554091560.76264	1554091570.77351	0.0530099869
18	1554091646.36245	1554091636.30388	1554091646.32124	0.0412099361
19	1554091721.95576	1554091711.89861	1554091721.90741	0.0483500957
20	1554091797.55627	1554091787.4974	1554091797.5077	0.0485699177
21	1554091873.14775	1554091863.08645	1554091873.09959	0.0481598377
22	1554091948.70667	1554091938.65154	1554091948.66079	0.0458800793

23	1554092024.30841	1554092014.26166	1554092024.27156	0.0368499756
24	1554092099.87953	1554092089.82066	1554092099.82925	0.0502798557
25	1554092175.46025	1554092165.40143	1554092175.40846	0.051789999
26	1554092251.07097	1554092241.01536	1554092251.02659	0.0443799496
27	1554092326.70357	1554092316.65063	1554092326.65925	0.0443198681
28	1554092402.28434	1554092392.2232	1554092402.23558	0.0487599373
29	1554092477.83592	1554092467.77899	1554092477.78809	0.0478301048
30	1554092553.37915	1554092543.3145	1554092553.3276	0.0515499115

All value		**No outliers**	
Average:	0.4148056269	Average:	0.0481163859
Median:	0.0484600067	Median:	0.0482549667
Fastest:	0.0368499756	Fastest:	0.0368499756
Slowest:	5.6293098927	Slowest:	0.0572497845
Standard Deviation:	1.3956489072	Standard Deviation:	0.0044948944