POLITECNICO DI MILANO

DIPARTIMENTO DI ELETTRONICA, INFORMAZIONE E BIOINGEGNERIA

Corso di Laurea Magistrale in Ingegneria Informatica

Towards Virtual Machine Consolidation in OpenStack

TESI DI LAUREA DI

Lorenzo AFFETTI
Matr. 799284

Giacomo BRESCIANI
Matr. 804979

RELATORE

Prof. Sam Jesus
Alejandro GUINEA
MONTALVO

Anno Accademico 2013/2014

Abstract

When writing code for a cloud computing software platform testing is a must. Unit testing and integration testing can help the developer know if his/her code will cause bugs in the overall system. However, cloud platforms are very complex pieces of software, with many many “moving pieces”. There are times when we will want to write code to change the system’s behavior —for example, to include new Virtual Machine placement or consolidation techniques. In these cases it is not enough to hunt for bugs. We need to know how the new code will behave within the system as a whole. To do this we need to be able to run complex simulations, and to collect and analyze the obtained results. However, it is difficult for a developer –who is often provided with limited amount of hardware resources– to experiment code in an entire cloud system which needs tens of physical machines to be hosted. Virtualizing the system is thus compulsory.

Our solution is aDock, a modular system that leverages Docker’s lightweight virtualization techniques and OpenStack –our reference open-source cloud computing software platform– to allow users to deploy extremely lightweight cloud systems, to run simulations, to store the system’s output, and to display it in real-time thanks to a friendly user interface.

As further contribution we developed a Virtual Machine Consolidation service for OpenStack, and tested it with four different consolidation algorithms. Virtual Machine Consolidation is one of the topics of primary importance in nowadays cloud systems. Consolidation is supposed to be an intelligent and efficient strategy for resource allocation, in order to make the most of available hardware and, thus, save energy. Energy saving, in fact, has become an urgent and important problem in data centers due to their growing energy greediness.

In order to evaluate our solution we used aDock to deploy an OpenStack system, and to benchmark and compare the proposed consolidation algorithms. Thanks to aDock we were able to deploy the system in a reasonable time — even on our laptops. The simulation results demonstrated that the four consolidation algorithms brought various degrees of improvements to the system’s resource allocation.

Sommario

Quando si scrive codice per una piattaforma software di cloud computing, il *testing* è di primaria importanza. Lo sviluppatore può utilizzare i test di unità e di integrazione per valutare la bontà del suo codice. Tuttavia, i software cloud possono essere molto complessi. Talvolta, inoltre, lo sviluppatore vorrebbe scrivere del codice che influenzi il comportamento dell'intero sistema e non di un solo componente. In questi casi il semplice *testing* non è sufficiente: lo sviluppatore ha bisogno di sapere come il nuovo codice si comporterà all'interno del sistema. E' quindi necessario eseguire simulazioni complesse ed analizzare i risultati ottenuti. Tuttavia, può essere difficile per lo sviluppatore, spesso sprovvisto di grandi quantità di risorse hardware, sperimentare codice in un intero sistema cloud, il quale è solitamente ospitato da decine di macchine fisiche. La virtualizzazione del sistema è quindi d'obbligo.

La nostra soluzione è aDock, un sistema modulare che sfrutta Docker per le sue tecniche di "virtualizzazione leggera" e OpenStack come software open-source di cloud computing di riferimento. aDock, in questo modo, consente agli utenti di avviare sistemi cloud estremamente leggeri, eseguire simulazioni, salvare i dati in uscita e mostrarli in tempo reale grazie ad un'interfaccia utente.

Come ulteriore contributo è stato sviluppato un servizio di consolidamento di macchine virtuali per OpenStack testato con quattro diversi algoritmi di consolidamento. Il consolidamento è, ad oggi, uno dei temi di primaria importanza nei sistemi cloud. Si tratta dello sviluppo di strategie per l'allocazione delle risorse che garantiscano l'utilizzo ottimale dell'hardware disponibile ai fini del risparmio energetico, problema sempre più rilevante nei *data center*, data la loro incessante crescita.

Al fine di valutare la nostra soluzione abbiamo usato aDock per lanciare un sistema OpenStack e confrontare gli algoritmi di consolidamento proposti. Grazie ad aDock siamo stati in grado di avviare il sistema in un tempo ragionevole, anche su computer portatili. I risultati delle simulazioni hanno dimostrato che i quattro algoritmi di consolidamento portano ad un notevole miglioramento nell'allocazione delle risorse del sistema.

Table of Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

The rapid growth of cloud services –in the past years– has driven a steep rise in the number of massively-scaled data centers. As a consequence energy consumption of data centers has become an urgent and important problem, as the power they need has reached the 1.3% of the world's total in 2010[3]. This is why an intelligent and efficient strategy for resource allocation is critical, to try to make the most of the available hardware.

Within an Infrastructure as a Service (IaaS) one way to achieve this goal is to try to minimize the number of running servers, while maintaining all the requested virtual machines running and available. This is commonly done by placing VMs intelligently on available servers. Exploiting this solution, it is possible to ensure that the data center is "filled" in a consistent way, avoiding under-allocated resources.

The problem with this solution is that it does not cover the cases in which VMs are deallocated from the hosting hardware. In these cases, in fact, the system could reach a state in which various –or all– the data center servers are used inefficiently, leaving some of them under-utilized and consuming more power because some of them may be potentially turned off.

To address this problem it is possible to periodically *consolidate* the arrangement of VMs within the data center, migrating them from under-utilized servers to servers which can host them; thanks to these migrations it might

be possible to "empty" the under-utilized machines and take them into an energy-saving power state, e.g., as a deep-sleep. As illustrated in the State of the Art (see Chapter 3), during the past years VM consolidation has gained more and more attention from the community, and a lot of algorithms and techniques have been proposed to address it.
Despite it being a very interesting approach to power saving, we lack solid VM Consolidation implementations, especially in non-proprietary IaaS; in fact, most of the solutions from the state of the art are theoretical —without practical tests in real environments. Take OpenStack, for example. It is the most important and used open-source IaaS solution, yet it doesn't provide any official implementation of the concept of VM consolidation.

In this thesis we decided to try to apply the concept of VM consolidation to OpenStack, since it is the reference platform for IaaS in the open-source world and it has a large and active community. More specifically, our goal was to implement a new OpenStack module that would allow us to "plug-in" and test a number of consolidation algorithms, and to see their impact on a real cloud system. At the beginning, however, we faced the problem of how to run, test and benchmark our code in an OpenStack environment. In order to deal with aspects like Scheduling, Virtual Machine Placement, and Server Consolidation, we needed a highly configurable system that would allow us to run simulations and benchmarks to evaluate the soundness of our solutions. Unfortunately we had limited server hardware, and could not construct a realistic testbed.

By looking at the state of the art we found that we were not alone; limited hardware is indeed a common barrier to experimenting with cloud infrastructure. Although OpenStack can be used to create testbeds, it is not uncommon in literature to find works that are plagued by unrealistic setups that use only a handful of servers.
For these reasons we decided to shift the focus of our work from VM consolidation algorithms to the implementation of a set of tools that would enable us to easily setup a cloud test environment. We needed a quick and easy way

to install and deploy code into a realistic experimentation environment, for example, to test a new consolidation algorithm. Moreover the experimentation environment would have to be lightweight, in order to allow it to be used with limited hardware resources (e.g., on a single developer workstation), and highly configurable to meet the needs of different situations.
However, setting up a testbed is necessary but not sufficient. One must also be able to create repeatable experiments that can be used to compare one's results to baseline or related approaches from the state of the art. We needed a way to automatically simulate, in a repeatable way, the workload generated from user applications that normally run on a cloud environment. Moreover we realized that it would be very useful to show real-time data of the simulations to analyze the behavior of the system in different configurations and analyze and compare the data collected.

Our solution to these problems was to develop aDock, a suite of tools for creating cloud infrastructure experimentation environments that are lightweight, sandboxed, and configurable. Our goal is to provide developers, sysadmins, and researchers a simple solution through which they can easily access a fully functional cloud installation of OpenStack.

aDock is made up of four main components: FakeStack (see 4.3), Oscard (see 4.4), Bifrost (see 4.5.1), and Polyphemus (see 4.5.2). FakeStack allows the user to manage the deployed OpenStack system, making it possible to configure and install one with the minimum effort. Oscard allows the user to configure and run repeatable simulations through command-line tools, as well as to store the experiments' outputs on Bifrost, i.e., our simulation database. Finally, Polyphemus allows the user to follow a simulation's progress in real-time, and to analyze relevant data such as the environment's resource utilization.
FakeStack, Oscard, and Polyphemus take advantage of Docker (see section 3.3.4 for more details), an open platform that exploits Linux containers to virtualize Operating Systems and Applications on a host; it allowed us to keep the overall solution lightweight, as well as to keep deployment quick thanks to

its low resource requirements, especially compared to more traditional Virtual Machines.

As a further contribution we used aDock to create a Virtual Machine consolidation module for OpenStack. The module provides extension points that make it easy to “plug-in” any consolidation algorithm we may want to test. To do this the module provides an abstract view of the experimentation environment and of its resource usage, allowing the more inexperienced developer to test his/her algorithms without having to fully understand OpenStack’s complex internal behavior.

In order to prove the soundness of our solution, we implemented four different consolidation algorithms: Random, Genetic, Genetic “best”, and Holistic; see 5.2 for more details. Using aDock, we were able to deploy a “1 + 5” OpenStack system —that is one containing one *controller node* and 5 *compute node*s. The compute nodes are the ones that actually host virtual machines. The controller node, instead, manages all of them.
This was achieved in a reasonable time on a laptop with very limited hardware resources —without compromising its usage. On a moderate Server machine we were even able to achieve a “1 + 42” configuration. All consolidation algorithms brought a significant improvement to OpenStack’s resource usage. In the bast case we were able to achieve a 14% increase in vCPU usage, a 20% increase in RAM usage, a 1.5% increase in disk usage, and a decrease of about 30% active nodes. Both aDock and our Consolidator module open the door to a lot of interesting future work. For what concerns aDock, Oscard will continue to grow to become a more and more realistic solution for running simulations. For example, we could exploit data from the state of the art to extract more realistic ratios between user actions (e.g. creating vs destroying Virtual Machines), user action density over time, and different application-specific virtual machine workloads (e.g. web applications, compute intensive tasks, etc.).

We would also like to make FakeStack more *modular*. Each OpenStack service could be dockerized into a single container, thus, making FakeStack even more flexible in terms of possible experimentation environment configurations. Up until now, more OpenStack services run in a single Docker container, making it harder to relize some specific OpenStack architectures. We are also interested in providing "ready-to-go" compositions of containers, which would make it even easier for the user to deploy an entire OpenStack system.

Regarding Virtual Machine consolidation in OpenStack, we believe our consolidator can provide a valid testbed for comparing approaches from literature (see sections 3.2.3 and 3.2.4). In the future we might be inetersted in providing a detailed survey of existing approaches, and base it on our consolidator module.

Chapter 2

OpenStack and DevStack

In this chapter we are going to present OpenStack and DevStack; we shall provide a brief overview and focus on the components and aspects that most concern the topics of our thesis to provide the reader with some key informations useful to the understanding of following chapters.

2.1 OpenStack

OpenStack is an open-source cloud computing software platform that provides a complete IaaS solution for public and private clouds. Founded by Nasa[1] and Rackspace Cloud[2] in 2010 OpenStack is now one of the biggest open-source projects with more than twenty thousand people working on it and more than twenty million code lines. It is a cloud operating system that controls large pools of compute, storage, and networking resources throughout a datacenter (with the possibility of controlling them trough a dashboard) and enabling enterprises and service providers to offer on-demand computing resources.
One of its main strengths is its modularity, which provides the flexibility needed to design different configurations for different cloud environments; its core components are:

[1] `www.nasa.gov` (2015)
[2] `www.rackspace.com` (2015)

Compute The service called `Nova` is the primary computing engine and it is used to deploy and manage large numbers of Virtual Machines.

Storage The storage platform, divided in Object Storage (`Swift`) and Block Storage (`Cinder`).

Network The service called `Neutron` offers Networking as a Service

Dashboard `Horizon` that is a dashboard provides users with a graphical user interface to access, provision, and automate cloud-based resources.

Shared Services Other services, that makes it easier to manage the IaaS, such as the Identity Service (`Keystone`), the Image Service (`Glance`), Telemetry (`Ceilometer`), Orchestration (`Heat`) and others.

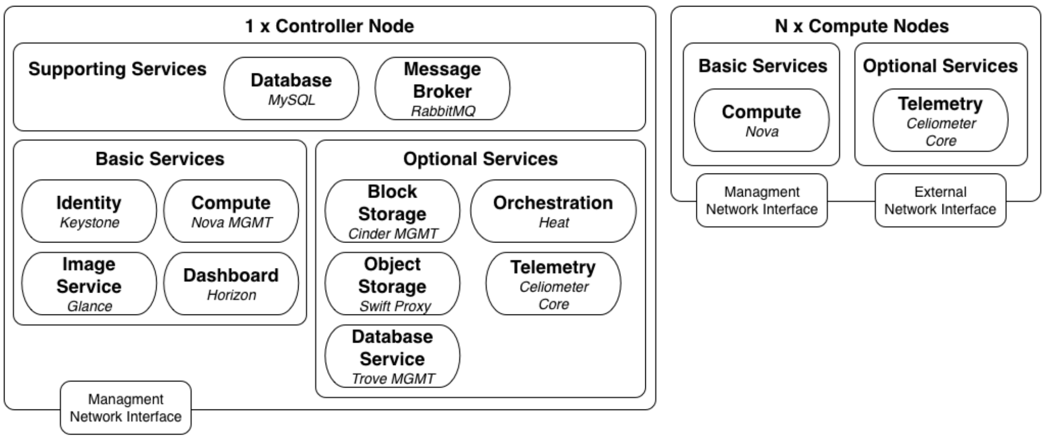


Figure 2.1: A “1 + N” OpenStack configuration

We have decided to focus on the creation of “1 + N” installations of OpenStack; these are composed of one *Controller* node and N *Compute* nodes with legacy networking. Legacy networking refers to a basic solution in which we do not deploy `Neutron` but we exploit `nova-network`, a `Nova` service described in paragraph 2.1.1. The figure 2.1 on page 8 provides a high level view of all the OpenStack services, both basic and optional, that have to be installed both on the Controller and the Compute nodes to setup a “1 + N” configuration.

The Controller node is responsible for globally managing the cloud operations. It runs the user `Identity` service, the Virtual Machine `Image` service, the management portion of the `Compute` service, and a *Dashboard* through which the users can request the creation of new Virtual Machines. Optionally, the node can run the management portions of the `Block`, `Object`, and `Database Storage` services, and the `Telemetry` and `Orchestration` services. The Controller node also runs a series of supporting services (i.e., the `Database` and `Message Broker` services). Each of the N basic Compute nodes, on the other hand, runs the `Compute` service and optionally the `Telemetry` service.

2.1.1 Nova

OpenStack Compute module, `Nova`, is the core of OpenStack; it takes care of deploying and managing Virtual Machines, by placing them on physical machines, letting them communicate, storing their informations on an SQL database, and offering a set of HTTP managed APIs and a command-line client. In the next paragraphs we will briefly describe the three `Nova` sub-modules that are relevant for our work.

Nova-compute The `nova-compute` is the sub-module that takes care of booting, resizing, live-migrating and destroying the Virtual Machines that are running on the physical servers, and of letting them communicate with the hypervisor.
Hereunder are reported four examples of the main commands (also accessible through the Nova APIs) used to boot, resize, destroy and live-migrate Virtual Machines:

- `$ nova boot --flavor <flavor> --image <image>`
- `$ nova resize --flavor <vm> <flavor>`
- `$ nova delete <vm>`
- `$ nova live-migration <vm> <host>`

Nova-network `Nova-network` is the basic network management module of OpenStack. It i included directly in `Nova`. Unlike `Neutron`, which can virtualize and manage both layer 2 (logical) and layer 3 (network) of the OSI network, `nova-network` only provides layer 3 virtualization and has some limitations on the network topology.
However `nova-network` is still supported by OpenStack and in powerful enough to support a "1 + N" configuration. It also streamlines the installation process as it avoids having to install another service (`Neutron`) and its dependencies.

Nova-scheduler `Nova` uses the `nova-scheduler` service to determine how to dispatch compute requests. It is used, for example, to determine on which host a Virtual Machine should launch. The system administrator can modify and configure the `/etc/nova/nova.conf` configuration file to adjust the criteria under which the `nova-scheduler` will place the Virtual Machines. The process of placing Virtual Machines on the most suitable host is divided in a *filtering* step, in which a list of candidate hosts is generated, and a *weighing* step in which the list is ordered according to the selected criteria and the best host is chosen.

2.1.2 Fake Drivers

To deal with situations in which the compute nodes are not physical machines that will host real Virtual Machines, but they have to be "fake" in the sense that they don't host a real hypervisor, OpenStack offers a module called `nova.virt.fake` that allows developers, that don't have real hardware, to test `Nova` code on compute nodes without a real hypervisor such as *libvirt*. When exploiting this solution Virtual Machines, are mere python objects; in such a way the Virtual Machines are not really spawned but simply stored in the database. However `FakeDriver` mimics the correct behavior of a real hypervisor, allowing one to test the rest of the `Nova` running flow.

This module is not configurable and by default a `FakeDriver` offers 1000 VCUs, 800000 MB of RAM, and 600000 GB of Hard Disk.

2.2 DevStack

DevStack[3] is a set of scripts and utilities to quickly deploy an OpenStack cloud environment and it is freely available on GitHub[4].
DevStack allows developers and system administrators to automate the process of installing OpenStack on a server reducing it to a simple command for every installation.
The services that are configured by default are Identity (`Keystone`), Object Storage (`Swift`), Image Storage (`Glance`), Block Storage (`Cinder`), Compute (`Nova`), Network (`Nova`), Dashboard (`Horizon`) and Orchestration (`Heat`). The main script is `stack.sh`; it does all the works, installing and configuring all the services set by the user.
All the required configurations, such as the Git repositories to use, the services to enable or the OS images to use, can be achieved overriding default environment variables (found in `stackrc`) through file `local.conf`. This is achieved with a `localrc` section, as shown below:

```
[[local|localrc]]
ADMIN_PASSWORD=secrete
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD
SERVICE_TOKEN=a682f596-76f3-11e3-b3b2-e716f9080d50
# ...
ENABLED_SERVICES=n-cpu,n-api,n-net
# ...
```

Listing 2.1: Sample of a `local.conf`

[3]More info at `docs.openstack.org/developer/devstack`
[4]`github.com/openstack-dev/devstack`

The environment variable `ENABLED_SERVICES` is used to define the service to run: in listing 2.1 the `Nova` services to install, in a simple compute node installation, are `nova-compute`, `nova-api`, `nova-network`. By running the script `tools/install_prereqs.sh` it is furthermore possible to install all the dependencies required by the configured services.
Other useful scripts provided by DevStack are `unstack.sh`, that allows to stop everything that was started by `stack.sh`, and `clean.sh` that tries to remove all the traces left by the OpenStack installation performed by DevStack.

Chapter 3

State of the Art

3.1 Introduction

As described in the Introduction (1) our thesis has two different topics. The first is the development of a system to create cloud infrastructure experimentation environments for developers and researchers. The second is the implementation of an OpenStack module to allow one to test consolidation algorithms to improve the resource allocation efficiency of the cloud infrastructure and, as a result, its energy efficiency.
For that reason this chapter is divided into two sections in which we are going to present the state of the arts of Cloud Test Environments and of Virtual Machines Consolidation.

3.2 Virtual Machine consolidation

At its most basic essence, cloud computing can be seen as a means to provide developers with computation, storage, and networking resources on-demand, using virtualization techniques and the service abstraction [4]. The service abstraction makes the cloud suitable for use in a wide variety of scenarios, allowing software developers to create unique applications with very small upfront investments, both in terms of capital outlays and in terms of required

technical expertise. Thanks to Cloud Computing, Internet software services have rightfully taken their place as important enablers in areas of great social importance, such as ambient assisted living [5], education [6], social networking [7], and mobile applications [8].
Managing a Cloud Infrastructure, however, presents many unique challenges. For example, there has been a lot of focus in the last few years on Virtual Machines Placement and Server Consolidation, given the role they play in optimizing resource utilization and energy consumption [1], [9]. Virtual Machine (VM) Placement [10], [11] defines how a cloud installation decides on which physical server to create a new virtual machine, when one is requested. Server Consolidation techniques [12], [13], on the other hand, allow a cloud provider to perform periodical run-time optimizations, for example through the live migration of VMs. The goal is always to desist from having too many under-utilized hardware resources given a specific workload, and to achieve this without compromising the quality of service that is offered to the cloud's customers.
Dynamic consolidation of Virtual machines is enabled by *live migration*, that is the capability to move a running Virtual Machine from one physical hosts to another with no downtime and no disruptions for the user. Thanks to dynamic Virtual Machine consolidation it is possible to minimize the number of active hosts, and to remove Virtual Machines from hosts when they become overloaded therefore avoiding performance degradation.

With regard to Virtual Machine Consolidation a lot of solutions, algorithms and techniques have been proposed in literature [14], [12], [13]; we decided to focus on four interesting papers described in sections 3.2.1, 3.2.2, 3.2.3 and 3.2.4. The section 3.2.5 is dedicated to the only attempt within the state of the art to apply Virtual Machine consolidation in the OpenStack world.

3.2.1 Genetic Algorithm

In the paper *Toward Virtual Machine Packing Optimization Based on Genetic Algorithm*[15] the authors explain how they modeled the problem of Virtual Machines consolidation as a bin packing problem and how they structured a Genetic Algorithm to deal with it. A Genetic Algorithm is a heuristic algorithm, i.e., a type of technique that is often used to address NP-hard problems such as the bin packing problem. A GA is a kind of machine learning that takes inspiration from the concept of evolution observed in biological environments, from which it borrows a lot of terms such as Chromosome, Mutation or Population.
The paper in question defines the concepts of a Genetic Algorithm for the Virtual Machine packing problem as follows:

Chromosome It represents a physical node, and in particular the list of hosted virtual machines.

Crossover They use a One-Point Crossover that randomly cuts two chromosomes and mix them. They also implement a repair function to fix the inconsistent children thus obtained.

Mutation They randomly exchange two positions between them.

Initial Population Generation They generate the initial population using a Minimal Generation Gap method.

Objective Function The unspecified objective function is said to be designed with parameters and weights in mind, such as SLA (Service level agreement) violations, number of active nodes, and number of migrations applied.

The experimentation environment and the simulation tests are not described in a detailed way and there are no data results to prove the soundness of the approach. Still, the idea of implementing a Genetic Algorithm to solve the consolidation problem is interesting, and possibly very efficient and useful; for these reasons we decided to take inspiration from it and implement a Genetic Algorithm, to be applied in an OpenStack test environment deployed with aDock, as described in section 5.2.2.

3.2.2 Holistic Approach

The paper *Energy Management in IaaS Clouds: A Holistic Approach* published during the IEEE Fifth International Conference on Cloud Computing in 2012 presents energy management algorithms and a holistic energy-aware Virtual Machine management framework for private clouds called Snooze.
The system architecture described by the authors (see figure 3.2.2 on page 18) is divided in three layers:

Physical layer It contains clusters of nodes; each is controlled by a Local Controller (LCs).

- *Local Controller* - They enforce Virtual Machines and host management commands coming from the GM (Group Manager, see below). Moreover, they monitor VMs, detect overload/underload anomalous situations and report them to the assigned GM.

Hierarchical layer It allows to scale the system and is composed of fault-tolerant components: Group Managers (GMs) and a Group Leader (GL).

- *Group Leader* - One GL oversees the GMs, keeps aggregated GM resource summary information, assigns LCs to GMs, and dispatches VM submission requests to the GM.
- *Group Managers* - Each of them manages a subset of physical hosts; they retrieve resource information and send commands, received by the GL, to the LCs.

Client layer It provides the user interface and it is implemented by a predefined number of replicated Entry Points (EPs).

The system addresses the scheduling problem both at the GL level, where VM to GM dispatching is done based on the GM resource summary information in a round-robin way, and at the GM level where the real scheduling decisions are made. In addition to the *placement policies*, which are applied when a new VM is requested, the work also supports *relocation policies*, which are called when overload or underload events arrive from LCs, and *consolidation policies*, which are called periodically according to one interval that is specified by the system administrator.
The paper proposes an algorithm for both overload and underload relocation policy. They both take as input the overloaded/underloaded LC along with its associated VMs and a list of LCs managed by the GM and output a Migration Plan (MP) which specifies the new VM locations.
The algorithm proposed for the consolidation follows an all-or-nothing approach and attempts to move VMs from the least loaded LC to a non-empty LC with enough spare capacity. LCs are first sorted in decreasing order based on their estimated utilization. Afterwards, VMs from the least loaded LC are sorted in decreasing order, placed on the LCs starting from the most loaded one and added to the migration plan. If all VMs could be placed the algorithm increments the number of released nodes and continues with the next LC. Otherwise, all placed VMs are removed from the LC and MP and the procedure is repeated with the next loaded LC. The algorithm terminates when it has reached the most loaded LC and outputs the MP, number of used nodes, and number of released nodes[1, p. 208].
In section 5.2.3 we describe how we implemented this algorithm in our system and the result obtained with our configuration.

3.2.3 Game Theory Approach

The paper *A Game Theory Approach to Fair and Efficient Resource Allocation in Cloud Computing* proposes a game theoretic resources allocation algorithm that considers the fairness among users and the resources utilization for both

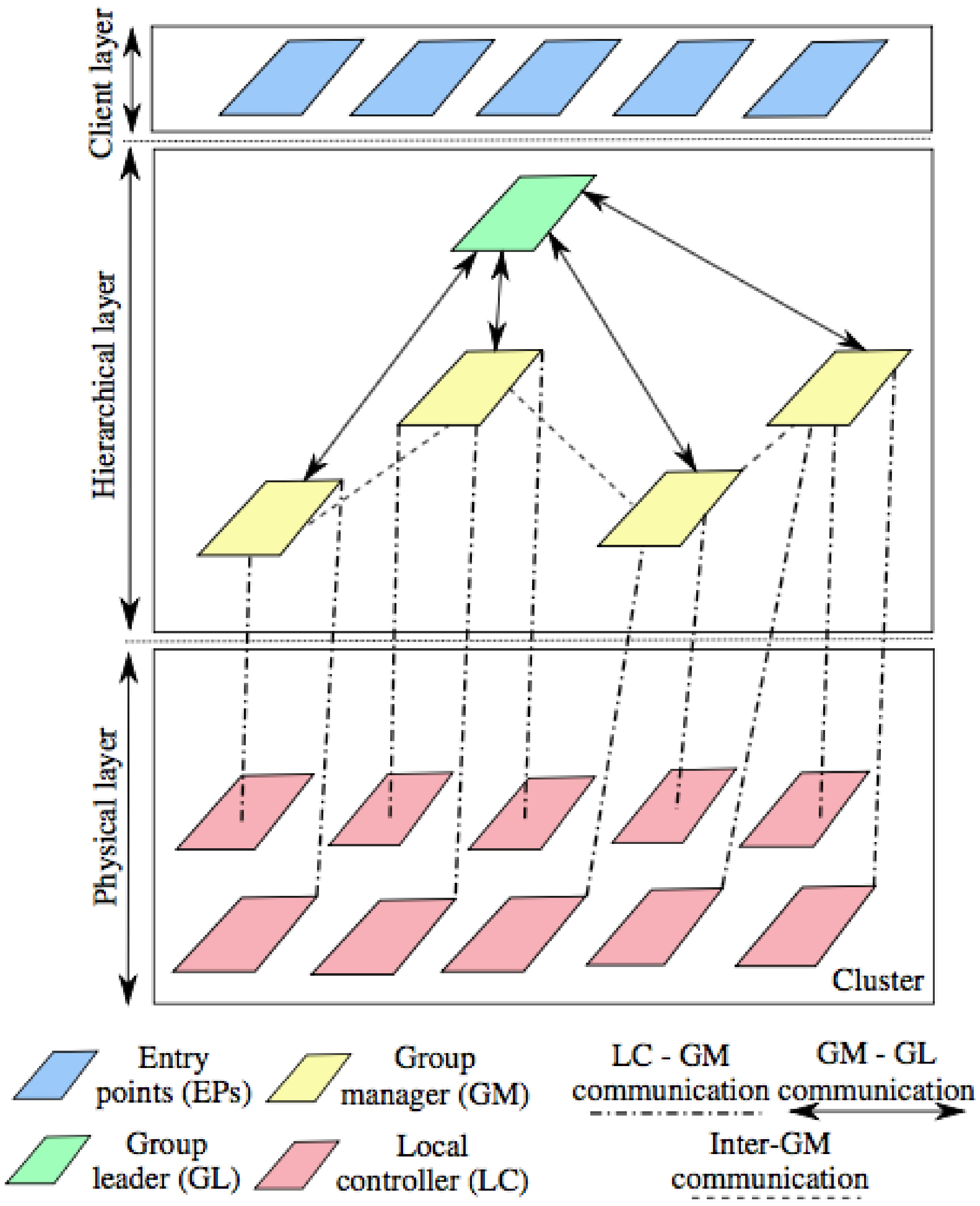


Figure 3.1: The Snooze architecture [1]

[16].

The four main components of the proposed cloud resource management system are:

CEM - Cloud Environment Monitor This component retrieves information like host names and IP addresses about physical servers, and monitors their statuses (starting, running, shutdown) and the consumption of CPU, memory, and disk storage.

RC - Register Center Every physical server in cloud data center should register its information to RC for connection and management.

IM - Infrastructure Manager It is responsible for deploying and managing the virtualized infrastructures, such as creating and releasing virtual machines.

CC - Control Center It is the control center to provide the most appropriate decision about resource allocating.

CEM monitors the statuses and resource consumptions for physical servers registered in RC. Once a new physical server joins the cloud, information like its MAC address and its IP address will be registered to RC. When a user sends a service request to the cloud, the resource requirements this request will be received by CC. CC makes an intelligent resource allocation decision based on the information collected by CEM. The allocation decision is executed by IM to manage the physical servers and place the virtual machines.[16, p. 3]

They experimented a FUGA (Fairness-Utilization tradeoff Game Algorithm) on a server cluster composed of 8 nodes and compared it to the Hadoop[1] *fair scheduler*[2]. They showed that it is possible to achieve an optimal tradeoff between fairness and efficiency compared with the evaluation of the Hadoop scheduler.

3.2.4 Multi-agent Virtual Machine Management

The solution presented in the paper *Multi-agent Virtual Machine Management Using the Lightweight Coordination Calculus* specifies the migration behav-

[1]A framework that allows for the distributed processing of large data sets across clusters of computers using simple programming models. `http://hadoop.apache.org`

[2]"Fair scheduling is a method of assigning resources to jobs such that all jobs get, on average, an equal share of resources over time.", `hadoop.apache.org/docs/r1.2.1/fair_scheduler.html`

ior of Virtual Machines within, and between cloud environments. It uses a Lightweight Coordination Calculus to provide an executable, declarative specification of an agent interaction model[17].

The proposed system is distributed between nodes; it doesn't have a central controller that could represent a single point of failure or a bottleneck. Agents located on the physical machines negotiate VM transfer between themselves, without referencing any centralized authority[17, p. 124].

The framework designed by the authors provides different types of interaction models by which it is possible to implement a wide range of algorithms and policies to support different situations.

3.2.5 Neat

OpenStack, at the state of the art, provides a comprehensive and efficient Virtual Machines Placement system. As described in section 2.1.1, it is part of the `nova-scheduler` module. However, with regard to Virtual Machines Consolidation, OpenStack does not include any official solution or plans to include it.

The only project that tried to bring Virtual Machine consolidation concepts to OpenStack is Neat[3]. It is defined as a framework for dynamic and energy-efficient consolidation of virtual machines in OpenStack clouds [2].

OpenStack Neat approaches the consolidation problem by splitting it in four sub-problems [2, p. 3]:

- Deciding whether a host is *underloaded.* In this case all Virtual Machines should be migrated from it, and the host should be switched to a low-power mode.
- Deciding whether a host is *overloaded.* In this case some Virtual Machines should be migrated from it to an other active host or a host should be reactivated to avoid violating the QoS requirements.

[3]`github.com/beloglazov/openstack-neat`

- Selecting the Virtual Machines to migrate from an overloaded host.
- Placing the selected Virtual Machines on an other active host, or on a reactivated one.

Figure 3.2.5 [2, p. 7] represents the architecture of OpenStack Neat: it is mainly composed by a *Global Manager* installed on the Controller node, and by a *Local Manager* and a *Data Collector* installed on every Compute node. The *Global Manager* is responsible for making global management decisions such as mapping Virtual Machine instances to hosts, and initiating Virtual Machines live migrations; the *Local Manager* makes local decisions such as deciding that the host is underloaded or overloaded and selecting Virtual Machines to migrate to other hosts; lastly the *Data Collector* is responsible for collecting Virtual Machines and hypervisors resource usage data and for then storing the data locally and submitting it to the central database, which can also be distributed.
One of the main characteristics of OpenStack Neat is that it is designed to be distributed and external to OpenStack, in fact it acts independently of the base OpenStack platform and applies Virtual Machines consolidation by invoking OpenStack's public APIs. For that reason it has to be installed separately from OpenStack, following the limited instructions present on the GitHub page of the project[4] and can not take advantage of tools like DevStack (see 2 on page 7) that automates the deploy and configuration of an OpenStack installation.

3.3 Cloud test environments

When provisioning a cloud we need to be able to test different environment configurations and algorithms, to analyze the behavior of new code that need

[4] `github.com/beloglazov/openstack-neat`

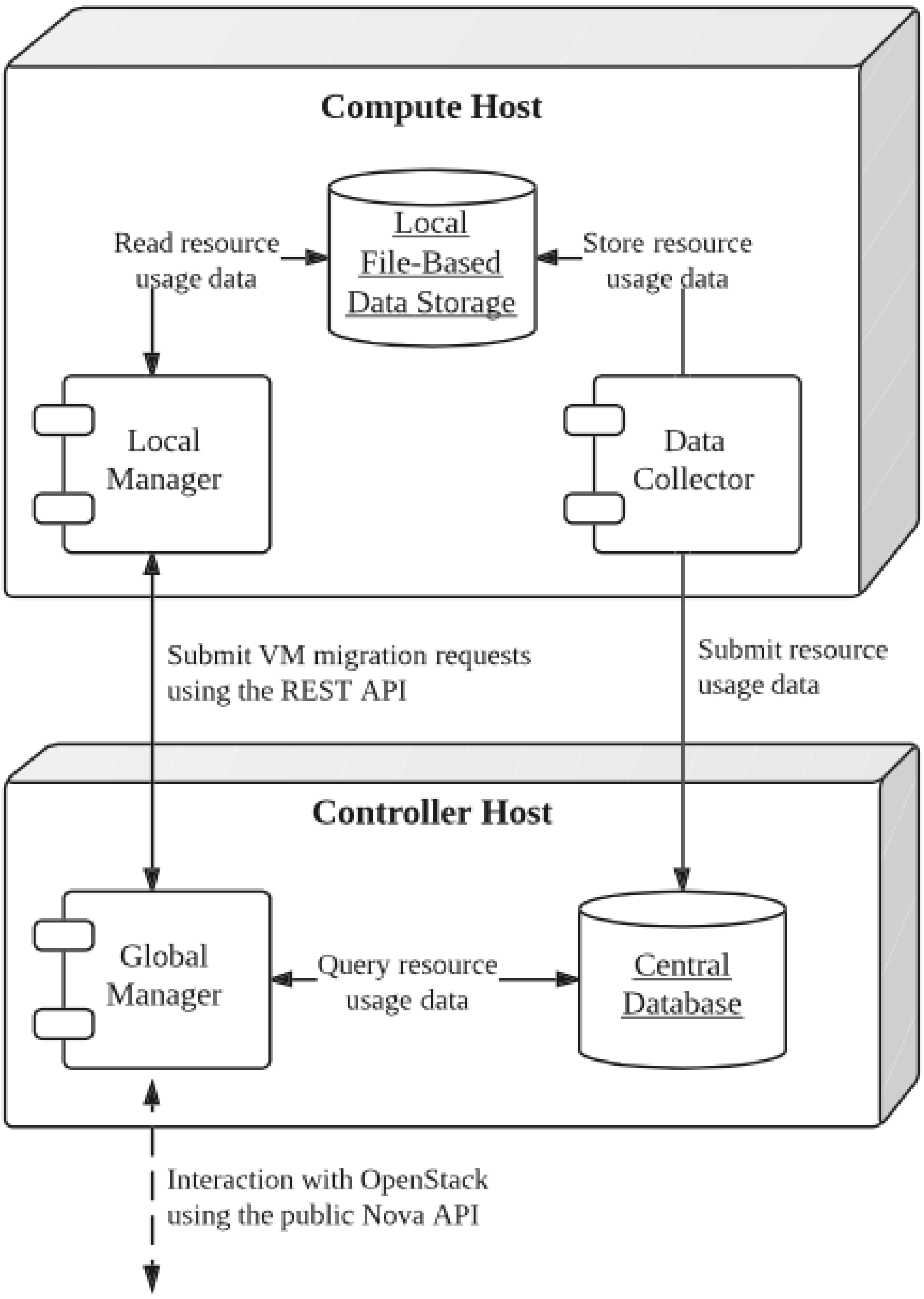

Compute Host
Local File-Based Data Storage
Read resource usage data
Store resource usage data
Local Manager
Data Collector
Submit VM migration requests using the REST API
Submit resource usage data
Controller Host
Global Manager
Query resource usage data
Central Database
Interaction with OpenStack using the public Nova API

Figure 3.2: The OpenStack Neat architecture [2]

to integrate with the environment, and to benchmark and collect data for research and experimentations. Unfortunately it can be expensive and complex to create and manage a cloud test environment in terms of time, resources and expertises, especially if the hardware resources like server machines or network infrastructures are limited. Fully understanding and handling an OpenStack installation is not easy, especially for non sysadmins like developers or researchers; it has a high learning curve and often a lot of time is needed to achieve the desired results.

There are some tools that reduce the impact of these complications are and make the process of setting up a cloud infrastructure experimentation environment easier and more manageable. The growing need of advanced system management have made configuration management tools, such as Chef and Puppet, have become increasingly mainstream. These tools provide domain-specific declarative languages for writing complex system configurations, allowing developers to specify concepts such as "what software packages need to be installed", "what services should be running on each hardware node", etc. More recently OpenStack has started collaborating both with Chef (see section 3.3.2) and Puppet (see section 3.3.3) to create new means to configure and deploy fully-functional OpenStack environments on bare-metal hardware, as well as on Vagrant virtual machines. The combination of a system management tool, like Chef or Puppet, and Vagrant can be used to setup a virtualized experimentation environment. However, these are complex sysadmin tools that require strong technical skills.

Below we present them and highlight their main features, as well as their strengths and weaknesses with respect to the topic of our thesis.

3.3.1 Vagrant

Vagrant[5] is a virtualization framework for creating, configuring and managing development environments, written in Ruby. It is a wrapper around virtualization software such as VirtualBox, KVM, VMware and could be used together with configuration management tools such as Chef and Puppet. Thanks to

[5] `www.vagrantup.com`

```
box      = 'trusty64'
url      = 'http://files.vagrantup.com/precise32.box'
hostname = 'customtrustybox'
domain   = 'example.com'
ip       = '192.168.0.42'
ram      = '2048'

Vagrant::Config.run do |config|
  config.vm.box = box
  config.vm.box_url = url
  config.vm.host_name = hostname + '.' + domain
  config.vm.network :hostonly, ip

  config.vm.customize [
    'modifyvm', :id,
    '--name', hostname,
    '--memory', ram
  ]
end
```

Listing 3.1: `Vagrantfile` example

an online repository [6] it is possible to automatically download a Vagrant Box and run it with a single command: `vagrant up vagrant-box-name`. It is also possible to create and configure custom Vagrant Box by simply writing a `Vagrantfile`: Provisioners in Vagrant allows one to automatically install and configure software in a Vagrant Box as part of the `vagrant up` process. Therefore it is easier to start with a base Vagrant Box, adapt it to your needs and eventually share it with other developers who can reproduce the same virtual development environment.
Vagrant is used together with configuration management software such as Chef and Puppet to create repeatable and easy to setup development and test environments that rely on Virtual Machines.

3.3.2 Chef

Description Chef[7] is a configuration management tool used to streamline the task of configuring and maintaining servers in a cloud environment.

[6] `www.vagrantcloud.com`
[7] `www.chef.io`

It can be integrated with cloud-based platforms such as Rackspace, Amazon EC2, Google Cloud Platform, OpenStack and others. It is written in Ruby and Erlang and uses a domain-specific language (DSL)[8] for writing configuration files called *recipes*. *Recipes* are used to define the state of certain resources[9], and everything that is required to configure the different parts of the system. They state what software should be installed (together with any required dependencies), services that should be run or files that should be written. Given a *recipe* Chef ensures that all the software is installed in the right order and that each resource state is reached, eventually correcting those resources in a undesired state; *recipes* can be collected into *cookbooks* to be more maintainable and powerful. In addition Chef offers a centralized hub, called Chef Supermarket[10]; it collects a large number of *cookbooks* from the community that are freely downloadable.
A base installation of Chef comprises three main components: a `chef-server` that orchestrates all the Chef processes, multiple `chef-clients` found on all the servers, and the user workstation that communicates with the Chef Server to launch commands.
To simplify communication with the `chef-server` Chef provides a command-line tool called Knife that helps users manage nodes, *cookbooks* and *recipes*.

Chef and OpenStack Chef and OpenStack can be combined and used together in different ways, many of which have a different goal compared to our thesis. It is possible, in fact, to deploy and manage a production OpenStack installation running on multiple servers and supervised by a Chef Server (using the subcommand `knife openstack`) to control the OpenStack APIs through Chef and to instantiate new physical servers with a `chef-client` or turn some off (`knife openstack server create / delete`). In this situation you can achieve a “1 + N” OpenStack configuration. In this case the OpenStack ser-

[8]A programming language specialized to a particular application domain.
[9]A resource state is a combination of installed software, running services, and configurations.
[10]`supermarket.chef.io`

```
machine 'controller' do
  add_machine_options vagrant_config: controller_config
  role 'allinone-compute'
  role 'os-image-upload'

  chef_environment 'vagrant-aio-nova'
  file('/etc/chef/openstack_data_bag_secret',
       "#{File.dirname(__FILE__)}/.chef/encrypted_data_bag_secret")
  converge true
```

Listing 3.2: Recipe to run an "All-in-One" configuration (`aio-nova.rb`)

vices are predefined and you cannot configure an ad hoc configuration. It is also possible to have an "All-in-One" configuration, where all the OpenStack services are installed on a single node.
These configurations can be achieved with the help of Vagrant that will cover all the steps to install OpenStack on a virtual machine and configure all its services (excluded Block Storage, Object Storage, Metering, and Orchestration). Within the OpenStack chef-repo[11] there is a *recipe* to configure a VirtualBox virtual machine that will host and All-in-One installation. Here is a part of it:

Of course it is possible to setup a "1 + N" configuration using different `Vagrantfiles` to create and configure one VM for the Controller and N VMs for the Compute nodes. However it is unlikely to succeed in running a lot of VMs on the same host, especially if they contain a fully functional OpenStack installation, since a Virtual Machine typically requires a significant amount of resources to operate.

Pros and Cons Chef is a very powerful tool to create, manage and configure cloud environments, and it offers a lot of functionalities to structure the desired architecture. In combination with Vagrant can also be used to setup test environments for development or research purposes.
However, with regard to this last aspect, it has several limitations:

- *Performance*: because VMs are very resource greedy it is very difficult to achieve a "1 + N" configuration for development or research purpose on

[11]`https://github.com/stackforge/openstack-chef-repo`

a single machine. On the other hand the “All-in-One Compute” solution that allows a full OpenStack installation on a single Virtual Machine is very simplistic and doesn’t represent a real environment setting.

- *Lack of customization*: at the state of the art all of the described solutions install both the Controller node and the Compute node with a predefined set of installed services (in practice all the OpenStack service excluded Object Storage, Metering, and Orchestration are installed) so it is not possible to setup the environment with more or less services or new ones. In our case, in fact, we need to be able to decide which OpenStack services to install (for example we don’t install OpenStack Network as a Service module, `Neutron`), as well as to implement a new one and install it (see chapter 5 regarding our Consolidator service).

3.3.3 Puppet

Description Similarly to Chef (described in section 3.3.2) Puppet[12] is a configuration management system that allows you to define the state of a cloud infrastructure, which it will then automatically enforce.
Puppet uses a declarative model where one defines the resource states; its manifest files are written in a Ruby-like DSL. Configuration files are enclosed in *modules*, self-contained bundles of code and data that are easy to share and reuse. There are a large amount of them on the Puppet Forge[13] repository.
Puppet is structured in a master-slave architecture: the master serves the manifests and the files, and the clients polls the master at specific intervals of time to get their configurations so that the master never pushes nothing to them.

Puppet and OpenStack As seen for Chef, Puppet can be very useful when dealing with OpenStack installation and maintenance. To configure and

[12]`www.puppetlabs.com`
[13]`forge.puppetlabs.com`

```
node 'control.localdomain' {
  include ::openstack::role::controller
}
```

Listing 3.3: Portion of a manifest file for a controller node

```
node 'storage.localdomain' {
  include ::openstack::role::storage
}

node 'network.localdomain' {
  include ::openstack::role::network
}

node /compute[0-9]+.localdomain/ {
  include ::openstack::role::compute
}
```

Listing 3.4: Portion of a manifest file for a compute node

deploy an OpenStack infrastructure with the help of Puppet one can download appropriate *modules* from Puppet Forge; this simplifies most of the operations such as OpenStack instances provisioning, configuration management and others. The module is `puppetlabs-openstack`[14]; using this module it is possible to deploy both a multi-node and an all-in-one installation. Compared to Chef, Puppet is a bit more flexible because it allows you to control more details about the OpenStack services that are to be installed on every node; for example, you can use the following instructions in the Puppet's manifest file of a node to achieve different results. In listing 3.3 we show a portion of a manifest file for a controller node, while in listing 3.4 we show it for a compute node.

Obviously, it is possible to configure multiple nodes to run in multiple Virtual Machines that are configured and launched with Vagrant and deploy the various OpenStack components with `puppetlabs-openstack`. This solution, is clearly difficult to achieve on a machine with a limited amount of resources; however also on a more powerful server machine this solution it is slightly feasible and scalable.

[14]`github.com/puppetlabs/puppetlabs-openstack`

Pros and Cons Puppet is an extremely powerful and mature tools for automated cloud infrastructure deploying: it streamlines the entire process and automates every step of the software delivery process.
However from our point of view we are more interested in knowing how it behaves when a single developer or a researcher needs to deploy a cloud infrastructure on a single machine with limited amounts of resources (a development workstation for example) and he/she has little sysadmin skills. With regard to this aspect Puppet used with Vagrant has some key limitations:

- *Performance*: A single Virtual Machine generally need a remarkable amount of resources, especially to host an OpenStack installation; for this reason it is very unlikely that one will be able to run on a single machine the number of Virtual Machines needed to deploy a realistic multi-node installation of OpenStack. Once again "all-in-one" solution is not sufficiently realistic, especially when testing algorithms or portions of code that involve multiple nodes.

3.3.4 Docker

Docker[15] is an open platform for developers and sysadmins to build, ship, and run distributed applications. Its core is the Docker Engine: it exploits Linux containers to virtualize a guest Operating System on a host avoiding the considerable amount of resources necessary to run Virtual Machines.
The main difference between the Docker solution and Virtual Machines solution lie in the way in which the hypervisor and the Docker Engine manage the guest Operating System. A Virtual Machine, as shown in figure 3.3.4 on page 30, hosts a complete Operating System including application, dependency libraries, and, more important, the kernel; the Docker Engine, on the other hand, runs as an isolated process in userspace on the host operating system and allows all the guest containers to share the kernel. Thus, it enjoys the resource isolation and allocation benefits of Virtual Machines, but is much more

[15] `www.docker.com`

portable and efficient; for our goals this aspect allows us to run at the same time a larger number of containers compared to what we are able to achieve with Virtual Machines and also to ship pre-built images of our modules.
To configure and build a container image you have to write a Dockerfile, that is a text document containing all the commands which you would have normally executed manually in order to take the container to the desired state, and then call `$ sudo docker build .` from the directory containing the file. The command `$ sudo docker run` will finally launch the container.
Docker offers an online platform called Docker Hub[16] where you can upload both Dockerfile and pre-built container images to streamline the sharing process.

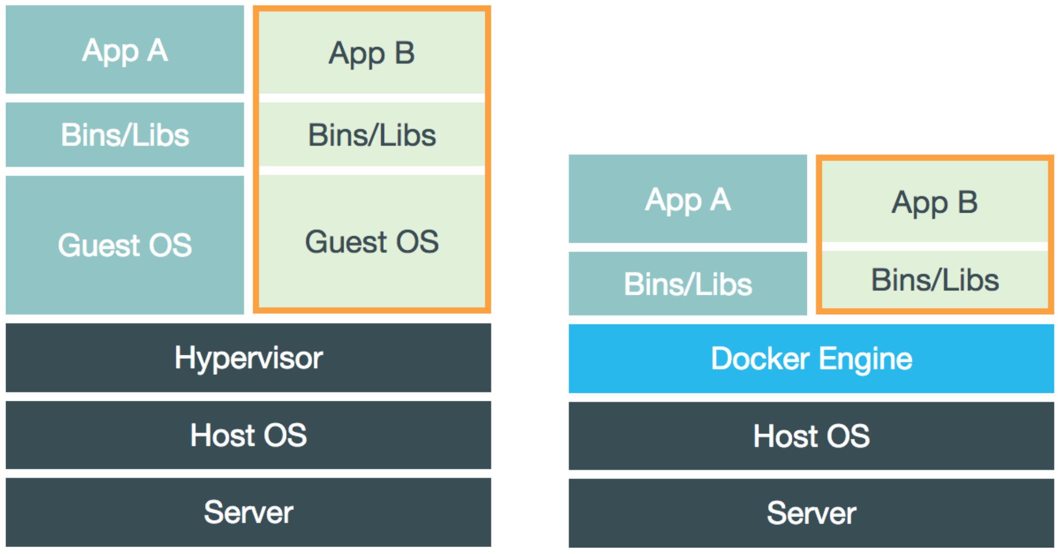


Figure 3.3: Hypervisor and Docker Engine

3.3.5 Dockenstack

One of the first attempts to create a cloud test environment based on OpenStack and Docker is Dockenstack[17]. It is an independent and not actively supported project, but is a good starting point to show the potential derived

[16] `hub.docker.com`
[17] `github.com/ewindisch/dockenstack`

from using Docker.
The project is basically composed of a Dockerfile and a bunch of scripts that will setup and configure an OpenStack installation using DevStack in a Docker container.
A pre-built image is available on Docker Hub, so with the command `docker run -privileged -t -i ewindisch/dockenstack` Docker will automatically download and run the container.
This made it a good solution for beginners wanting to learn OpenStack, but inadequate for advanced experiments, such as experiments regarding Virtual Machine placement and server consolidation algorithms.

Chapter 4

aDock

4.1 Our Solution, aDock

The lack of a uniform and standardized test environment for cloud systems brought us to develop aDock.
aDock is a suite of tools that lets the final user deploy a complete OpenStack system; run simulations against it; collect output data and view results on a friendly user interface.
We chose OpenStack as our reference cloud computing software platform, because it is open-source and because it is continuously evolving to keep up with the latest cloud standards.
Our intended users are OpenStack developers who need to run their code in a fully functional environment, and researchers who want to try their algorithms (e.g. about virtual machine placement or consolidation) on a complete cloud system to test out their behavior.

4.2 Requirements

In this section, we will identify both the functional and non-functional requirements for aDock. Functional requirements (see subsection 4.2.1) lead us to define the general architecture of the system proposed, aDock, which we show in figure 4.2.2 (see subsection 4.2.2). Non-functional requirements lead us to precise choices in technologies used to develop the system (see subsection 4.2.3).

4.2.1 Functional Requirements

FR1 *aDock should provide tools to deploy a complete environment.*
A user should be able to start and update OpenStack's nodes with a single command and to decide which services will be installed and started on each node and their internal configuration using a configuration file.

FR2 *aDock should provide a tool to run simulations.*
If the user puts his/her code into OpenStack he/she probably will need to run simulations and examine how the new piece of code behaviors in interacting within the rest of the system. Simulations should be configurable according to the user needs and repeatable.

FR3 *aDock should persistently store the simulations' output.*
Once a simulation has been run, it could be interesting to store its outputs in terms of generic metrics about the system, such as the average number of compute nodes active during the simulation, the average number of virtual CPUs used, and so on.

FR4 *aDock should provide a user interface.*
Simulation results should be displayed to the user in a friendly manner, using charts to give the user a glimpse of the current situation. Data representation should also be given in real-time.

4.2.2 aDock Modules

We decided to divide aDock into different components. Each component satisfies one or more requirements (see subsection 4.2.1). We give here a high-level description of each aDock's module and how it satisfies some of the requirements given.

FakeStack (see section 4.3) is the aDock module which provides the user with the tools specified in requirement 1. FakeStack provides the concept of "node" which, in its depth, is an Ubuntu based Docker container, shipped with OpenStack services dependencies. Starting a node is as easy as `$ run_node`. A node can be configured by means of a simple configuration file. Nodes are of two types, *controllers* and *computes*. Controller nodes are different from compute ones because they are shipped with *MySQL* and *RabbitMQ* installations.

Oscard (see section 4.4) is the simulation tool that satisfies requirement 1 and 2. Oscard is the aDock module which takes care of running repeatable and configurable simulations against an OpenStack system and to store outputs persistently. This module, by default, stores the aggregates of a simulation into a Firebase[1] backend. In our case, the backend is called Bifrost (see section 4.5).

Although Firebase provides an interactive user interface, data is displayed as *JSON* and is, therefore, not easily understandable and browseable. Polyphemus (see section 4.5) is the aDock module that takes care of displaying real-time simulation results in a friendly manner satisfying requirement 4.

aDock is a modular system where each component is configurable and has a precise purpose. FakeStack is employed to start nodes; Oscard runs simulations and collects aggregates on Bifrost; Polyphemus is the eye on the data that shows the user the obtained results. In figure 4.2.2 we highlight the general architecture of aDock.

[1]`https://www.firebase.com/`

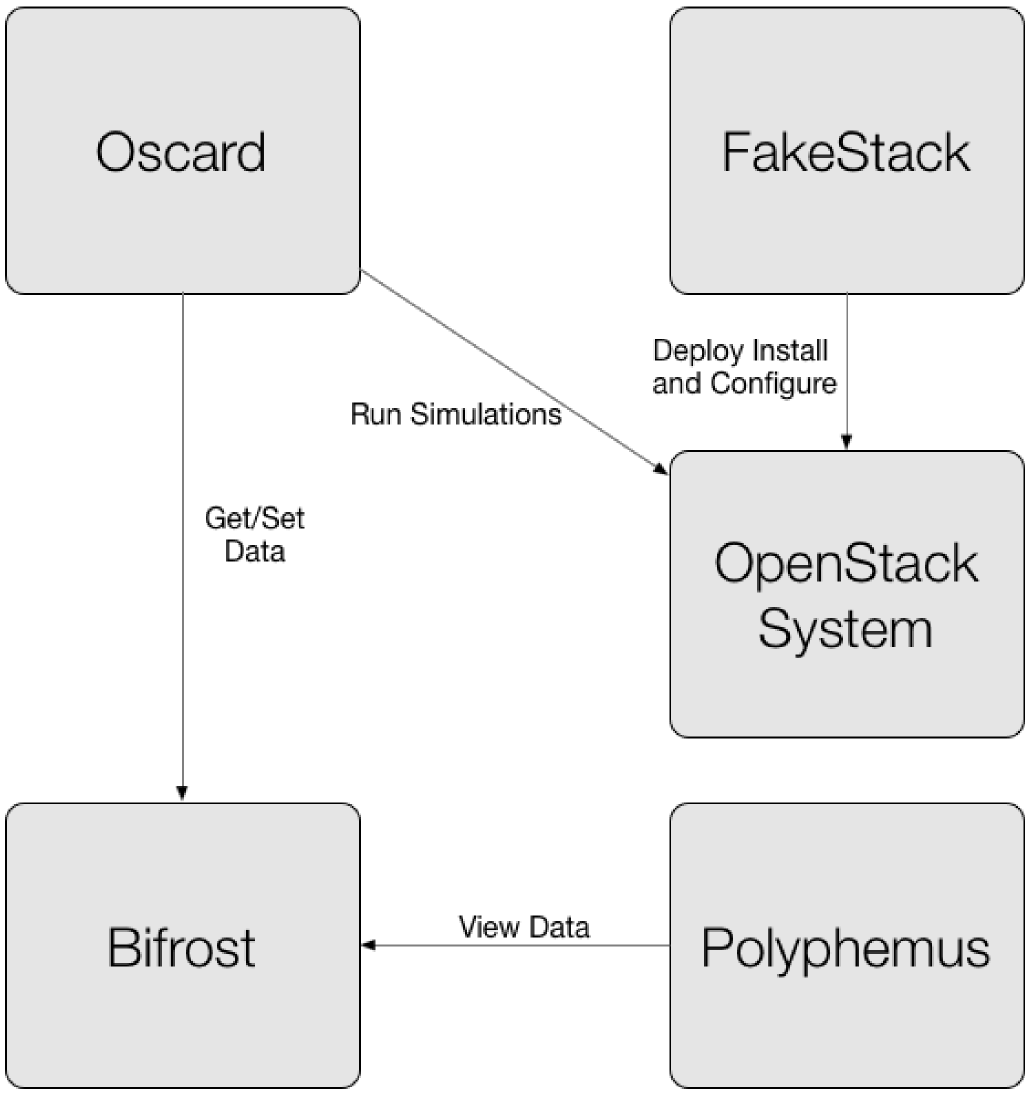


Figure 4.1: aDock's high level architecture

4.2.3 Non-Functional Requirements

aDock also has very strong non-functional requirements, to give a suitable testing environment to our stakeholders. In general we take leverage of Docker and DevStack and use their biggest strength. Docker gives us i) high speeds in running containers, ii) sandboxing by construction, and iii) makes aDock cross-platform. DevStack gives us great flexibility and configurability for what concerns OpenStack services.

NFR1 *Users should be able to choose which OpenStack's code version is running.*
Before booting the entire system the user should be able to choose if he/she wants to run OpenStack code from a precise code repository which is, in general, the better way to version and share code amongst developers. Speaking in Git[2] terms, a user could choose to run the most up to date code (which may be buggy) and so get the code from branch `master`, or maybe get a much more stable OpenStack version and get the code from branch `stable/juno`. The most interesting fact (and this is the scenario we have in our mind) is that the user could choose to fork an OpenStack service and see his/her code running nodes.

Solution All of this is achievable thanks to DevStack, which installs OpenStack services by cloning repositories from GitHub and running `python setup.py install`. By default, DevStack clones official OpenStack repositories from branch `master`, but it is possible to specify different repository URLs and branches for each of the OpenStack services by means of `local.conf` files.

NFR2 *aDock should be lightweight.*
Users often need to test algorithms that, by design, target the management and/or optimization of tens of physical servers. Since we can assume that not everyone will have that amount of resources, we believe that aDock should be as light-weight as possible. It should be possible to run aDock on limited hardware, potentially even on one's personal laptop. It is under this assumption that sandboxing becomes important; indeed, the experimentation environment should not have any sort of repercussions on the user's machine; we want the user to be able to build and tear down the environment with no consequences.

[2] `http://git-scm.com/`

Solution Docker is a virtualization system which relies on Docker containers which are much more lightweight than virtual machines[3][4]. Docker gives us, by construction, speed and lightness.

NFR3 *The experimentation environment should be highly configurable.*
Our primary goal with aDock is to provide a fast and easy way to create the experimentation environment. We believe that building a system which allows users to design the overall architecture of the cloud system is out of scope of this thesis, mainly because of the intrinsic high complexity and vastness of OpenStack's system itself. Up to now, as a proof of concept, we will focus on "1 + N" architecture, with 1 controller node and N compute nodes.

Solution The possibility to configure the system still remains in configuring OpenStack services in terms of their internal behavior. This is achieved, again, thanks to DevStack, which allows us to configure all aspects of OpenStack through its `local.conf` file. Each service can be configured in each of DevStack's installation phases. Each service, during installation, passes through **local**, **pre-install**, **install**, **post-config**, **extra** phases[5]. Configuring a service is as simple as adding few lines to `local.conf` file as shown in listing 4.1 on page 39.

NFR4 *aDock should allow users to run repeatable simulations.*
It is of paramount importance that users be able to compare their results with baseline approaches, as well as with related work from the state of the art. aDock should make it easy to compare an experiment's results with those of others on the same simulations.

[3]From Docker: "Containers boot 1000x faster than virtual machines; their disk and memory footprint are also much lower; and they work on virtually all current platforms" (see `https://www.docker.com/company/careers/?gh_jid=47837`).

[4]`http://devops.com/blogs/devops-toolbox/docker-vs-vms/`

[5]`http://docs.openstack.org/developer/devstack/configuration.html#local-conf`

```
... # DevStack configurations

[[post-config|\$NOVA_CONF]]
[DEFAULT]
verbose=True
logdir=/var/log/my-nova-logdir

# SCHEDULER
compute_scheduler_driver=nova.scheduler.MyMagicScheduler
# VIRT DRIVER
compute_driver=nova.virt.fake.MyAmazingFakeDriver
```

Listing 4.1: Adding per-service configuration to DevStack's local.conf file

Solution Oscard will take into account repeatability both giving the possibility to run the same simulation, at the same time, on multiple hosts, both using pseudo-randomization (see section 4.4).

4.3 FakeStack

FakeStack[6] is the aDock module that allows the user to manage *nodes*, the building blocks of an OpenStack system. Nodes are of two types: *controllers* and *computes*[7]. Both of them are Ubuntu-based Docker containers shipped with pre-installed software that satisfies most[8] of OpenStack's services dependencies. Both controller and compute nodes are configurable by means of simple configuration files 4.3.3.
FakeStack provides a set of scripts to handle node startup, service updating on live nodes and other features 4.3.2.

[6]`https://github.com/affear/fakestack`

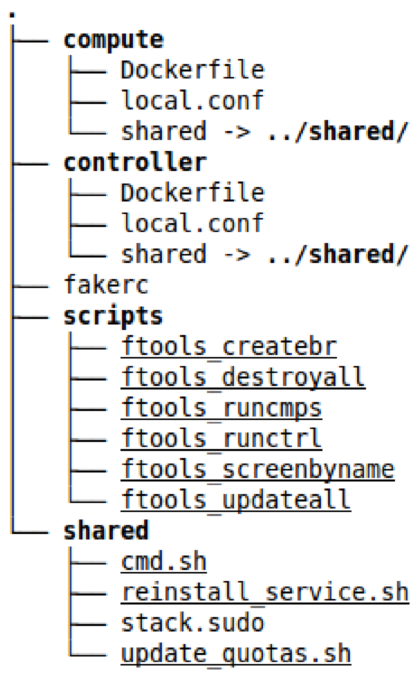
[7]In FakeStack, compute nodes, by default, are equipped with `nova.virt.fake.Fakedriver`. Thus, FakeStack compute nodes don't host a real hypervisor such as *Libvirt*. Virtual machines, in FakeStack, are mere python objects. This fact, doesn't influence user's choices. A user, in fact, can equip FakeStack's compute nodes with `nova.virt.libvirt.driver.LibvirtDriver` as long as he/she satisfies its dependencies (this brings to editing `Dockerfile` and rebuild node's image).

[8]main services as *Nova*, *Keystone*, *Glance* are actually supported.

4.3.1 Nodes

As already anticipated in requirement 4.2.3, we will focus on “1 + N” architectures. This architecture is characterized by 1 controller node that handles *N* compute nodes.
The main difference between a controller and a compute node is that the former contains a database (in our case, *MySQL*) and a message broker (in our case, *RabbitMQ*); both are compulsory for OpenStack’s controller nodes.

To understand how FakeStack really works, it is useful to examine its internal structure:



```
.
├── compute
│   ├── Dockerfile
│   ├── local.conf
│   └── shared -> ../shared/
├── controller
│   ├── Dockerfile
│   ├── local.conf
│   └── shared -> ../shared/
├── fakerc
├── scripts
│   ├── ftools_createbr
│   ├── ftools_destroyall
│   ├── ftools_runcmps
│   ├── ftools_runctrl
│   ├── ftools_screenbyname
│   └── ftools_updateall
└── shared
    ├── cmd.sh
    ├── reinstall_service.sh
    ├── stack.sudo
    └── update_quotas.sh
```

Figure 4.2: FakeStack’s file structure

As we can see in figure 4.3.1, nodes have two separated `Dockerfiles` (which makes them two different Docker containers), but they share a set of scripts (contained in the `shared` folder):

cmd.sh This is the script that will be run when the container starts (`docker run`). In algorithm 1 we explain its behavior in terms of pseudo-code.

Algorithm 1 `cmd.sh` behavior

```
if node is controller then
    set last IP in Docker bridge      ▷ assign static IP address to controller
end if
ping 8.8.8.8                          ▷ Check internet connection
if node is controller then
    start mysql
    start rabbitmq-server
end if
./stack.sh                            ▷ real OpenStack installation (using DevStack)
/bin/bash                             ▷ let the user work on the container
```

reinstall_service.sh This script allows the user to update a service on this node specifing its name (e.g. `nova`, `glance` and so on)

update_quotas.sh This script allows the user to enlarge *quotas* for the *tenant*[9] in use (in our case `admin`) to a very big amount. Its usage is justified by the fact that the user will probably spawn hundreds or thousands of virtual machines on his/her OpenStack nodes, and that, normally, standard quotas will prevent him from doing so. Enlarging quotas is an easy and fast way to allow the user to not worry about how many virtual machines he owns.

Once a node has been built, all of the shared scripts are copied to the file-system of the node.
When building a container, in fact, we specify in `Dockerfile` which files have to be copied into container's file system. The files specified are copied at build-time and subsequent changes on user's file system will not result in a modification at container's file system. *Data Volume*[10] is a feature of Docker's that allows modifications on files to be immediately applied to container's file system. `local.conf` file *is* a *Data Volume*, thus it is "bound" to the node's file-system, in order to avoid rebuilding at each modification.

[9] *Quotas* are limits on how much CPU, memory and disk space the tenant can use. *Tenant* is an OpenStack concept similar to Linux groups.

[10] `https://docs.docker.com/userguide/dockervolumes/#data-volumes`

When a node is started, `cmd.sh` will run, which in turn will run DevStack's `stack.sh` (see section 2.2). At that point OpenStack's installation starts.

4.3.2 Scripts

Once a user enters FakeStack's root directory he/she has to perform `source fakerc`. Executing this command all of the scripts contained in the `scripts` directory become available in `PATH`. The prefix `ftools_` is added to avoid conflicts in names.
Scripts for running and updating nodes leverage Linux screens[11]. Screens is a powerful tool to run detached shells from within another shell. This feature gives lots of advantages both in terms of ease of use and in running long running jobs via SSH.
All of aDock processing is confined into two different screens, `running` and `updating`. Thanks to this, the user will not have to open lots of shells, but focus on using only one; keeping it clean from computation and reattaching to aDock screens when needed. If the user wants to run a long running task (e.g. a very long simulation or lots of different, small simulations) on a remote server via SSH, he will not need to keep the SSH session open and wait for simulations to end; the screen session, in fact, will stay open (and so the processes within it, running) independently from the SSH connection.

We now list and describe the scripts contained in the `scripts` directory.

createbr Input: bridge name. Creates a bridge with CIDR 42.42.0.0/24; it takes the name passed as first argument by the user. This bridge is intended to be used by Docker, setting the option `-b <bridge_name>` into `/etc/default/docker`. Run this script before starting the system or editing the IP configuration in `cmd.sh`. In fact, `cmd.sh` will set the controller's IP to the last IP available in that precise network (42.42.255.254). After running this script, Docker service has to be restarted.

[11] `http://linux.die.net/man/1/screen`

destroyall Stops and destroys all OpenStack nodes.

runctrl Runs one controller node on a new window in screen `running`.

runcmps Input: `N`. Starts `N` compute nodes concurrently[12]. A new window (`cmps`) in screen `running` is created asking for operation confirmation. Once the opration is confirmed, nodes are started and a new window (`samplecmp`), attached to one of the compute nodes, is opened to show the user a sample node behavior and progress in OpenStack installation.

screenbyname Input: screen name. Reattachs to the screen named as given by the user, if it exists.

updateall Input: service name. Updates the service given by the user on all OpenStack nodes. All of the processing is performed into screen `updating`.

We will now list and describe the scripts "sourced" by `fakerc` (`ftools_` convention is always maintained).

runcmp Alias for `ftools_runcmps 1`.

build Input: `ctrl` or `cmp`. This script builds the node; regenerating it from a pure Ubuntu image. It is necessary to run this script only in case some of the files (apart from `local.conf`) have been modified.

attach Input: container ID. Attaches to a Docker container. Alias for `docker attach <container_id>`.

4.3.3 Configuration

Fakestack leverages the powerful configurable options of DevStack. Modifying `local.conf` files before starting a node, it is possible to change the enabled services (see listing 4.2) and their internal configuration (see listing 4.3).

[12]They are started as Docker daemons (`-d` option in Docker).

```
... # Other configuration options

# Enables:
# - Nova Compute
# - Nova API
# - Nova Network
ENABLED_SERVICES=n-cpu,n-api,n-net

... # Other configuration options
```

Listing 4.2: Choose OpenStack's enabled services

```
... # Other configuration options

[[post-config|$NOVA_CONF]]
[DEFAULT]
compute_driver=nova.virt.fake.MyFakeDriver

... # Other configuration options
```

Listing 4.3: Internal configuration of Nova

Every service is configurable in each of its installation *phases*, which are, for DevStack, **local**, **pre-install**, **install**, **post-config**, **extra**[13]. Configuring it is as simple as adding a `[[ <phase> | <config-file-name> ]]` line (e.g. `[[post-config|$GLANCE_CONF]]`) to `local.conf` file and add configuration options below.

Most important, it is possible to choose a different Git repository and Git branch for each of OpenStack's services enabled (see listing 4.4). DevStack, in fact, install services *cloning* those repositories and running `python setup.py install`[14].

Thanks to this important piece of configuration, a user can *fork* an OpenStack project; develop its code and use its new forked repository URL in DevStack's configuration.

In listing 4.5 we show a possible complete example of `local.conf` file for a compute node.

[13]For more information see `http://docs.openstack.org/developer/devstack/configuration.html#local-conf`

[14]It is the standard way to install *PyPI* packages. More information can be found at `https://wiki.python.org/moin/CheeseShopTutorial`

```
... # Other configuration options

NOVA_REPO=https://github.com/me/nova.git
NOVA_BRANCH=my-branch

... # Other configuration options
```

Listing 4.4: Change repository URL

```
[[local|localrc]]
FLAT_INTERFACE=eth0
MULTI_HOST=1
LOGFILE=/opt/stack/logs/stack.sh.log
SCREEN_LOGDIR=$DEST/logs/screen

NOVA_REPO=https://github.com/me/nova.git
NOVA_BRANCH=my-branch

DATABASE_TYPE=mysql

ADMIN_PASSWORD=pwstack
MYSQL_PASSWORD=pwstack
RABBIT_PASSWORD=pwstack
SERVICE_PASSWORD=pwstack
SERVICE_TOKEN=tokenstack

SERVICE_HOST=controller
MYSQL_HOST=controller
RABBIT_HOST=controller

NOVA_VNC_ENABLED=False
VIRT_DRIVER=fake

ENABLED_SERVICES=n-cpu,n-api,n-net

[[post-config|$NOVA_CONF]]
[DEFAULT]
compute_driver=nova.virt.fake.MyFakeDriver
```

Listing 4.5: Complete `local.conf` example for compute node

4.3.4 Example

In this section we provide an example on how to use FakeStack in a pseudo-code fashion.

Procedure 2 is comprehensive of real bash commands, aDock's commands and standard input to handle Linux screens. It refers to a user that wants to launch a "1 + 5" architecture from scratch. In this case we suppose that the user will start a "vanilla" OpenStack version, and so he/she doesn't need to modify any configuration files.

Algorithm 2 Launching a "1 + 5" architecture with aDock

```
git clone https://github.com/affear/fakestack
cd fakestack
source fakerc                     ▷ all aDock commands are now available
ftools_createbr docbr             ▷ "docbr" is the name of the new bridge
sudo nano /etc/default/docker ▷ adding -b docbr option to Docker's configuration
sudo service docker restart

ftools_runctrl
                                  ▷ waiting for controller to finish installation
ftools_runcmps 5
screen -R     ▷ attaching to the only screen active (running). Window is ctrl
CTRL+A N                                                  ▷ window is now cmps

enter y to confirm that a controller node is up and we want to start compute nodes

                                            ▷ wait for compute nodes to finish
CTRL+A P for two times                                    ▷ ctrl window
source openrc admin admin
nova service-list                           ▷ 5 compute nodes should be shown
nova boot --image cirros --flavor 1 samplevm      ▷ Spawns a new virtual machine
...
...                                        ▷ enjoy your OpenStack environment
```

4.4 Oscard

Oscard[15] is the aDock module that takes care of running simulations against one or more OpenStack systems and collecting their data outputs. Oscard has two main components, a server and a client (see 4.4.2). The two components don't need to be used on the same machine. This is why Oscard's dockerized version only runs the server part and waits for requests from the client.

The client part is the one that defines simulation running behavior. The user is supposed to use the client to actually start simulations by means of an executable file (`$ ./bin/run_sim`).

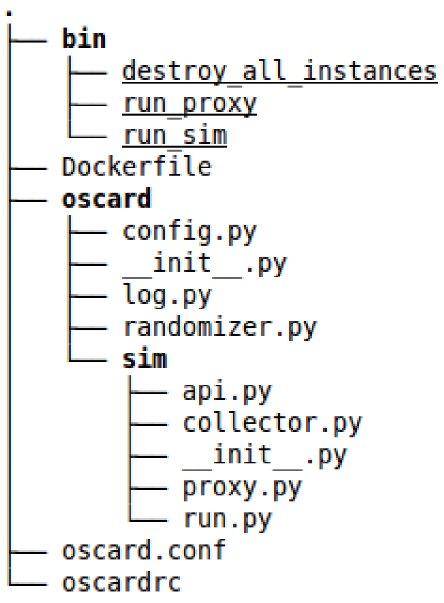
The server part is the *Proxy*, it literally waits for client requests and forwards them to OpenStack's controller node and stores simulation's outputs into Bifrost (see section 4.5.1).

Oscard is completely configurable from the `oscard.conf` file.

4.4.1 Modules

Oscard is composed of few modules (see figure 4.4.1); in this section we will explain in detail what each of them is up to.

[15]`https://github.com/affear/oscard`



```
.
├── bin
│   ├── destroy_all_instances
│   ├── run_proxy
│   └── run_sim
├── Dockerfile
├── oscard
│   ├── config.py
│   ├── __init__.py
│   ├── log.py
│   ├── randomizer.py
│   └── sim
│       ├── api.py
│       ├── collector.py
│       ├── __init__.py
│       ├── proxy.py
│       └── run.py
├── oscard.conf
└── oscardrc
```

Figure 4.3: Oscard's file structure

oscard.sim.api This module contains the APIs to interact with OpenStack's Nova. It contains two classes, `NovaAPI` and `FakeAPI`. `NovaAPI` provides the methods necessary to perform basic operations on Nova using OpenStack's official python clients: `keystoneclient.v2_0` and `novaclient.v1_1`.

- `init`: resets APIs random seed. If the option `random_seed` has been modified in `oscard.conf`, the new seed will be reloaded as well.
- `architecture`: Returns the system's architecture in terms of compute nodes and their resources (vCPUs, memory and disk).
- `active_services`: Returns active service and their number. For instance, if there are 10 compute nodes, it is very common to have 10 `nova-compute` services up. In this case, only one `nova-compute` service with count 10 will be returned.
- `snapshot`: Returns a snapshot of the system. The snapshot contains data about each compute node in use (a node which is host-

ing virtual machines), such as resources in use, and aggregate data about all active nodes (averages of resources usage).

- `create` : Spawns an instance of random flavor and returns its ID.
- `resize` : Resizes a random active instance to a random flavor (different from its actual one) and returns its ID.
- `destroy` : Deletes a random instance.

`FakeAPI` class, mimics `NovaAPI` 's behavior but it doesn't involve an OpenStack controller. It was developed only for testing purpose.

oscard.sim.collector This module exposes the `BifrostAPI` class; this API gives a way to interact with the Firebase backend for data storage. An instance of `BifrostAPI` is obtained in the `oscard.sim.run` module, and is used to store the data obtained during the simulation.

oscard.sim.proxy This module contains the API to communicate with the proxy (`ProxyAPI`). The class basically mirrors the methods contained in `oscard.sim.api.NovaAPI` . A `NovaAPI` object is obtained when the module is started, to which calls are *delegated*. It is this module that, if run from module `__main__` , starts a WSGI server (powered by Bottle[16]) and waits for GET and POST requests.

oscard.randomizer This module is a wrapper for python's `random` module. It provides `get_randomizer` function which returns a new `random.Random` object initialized with the same seed as specified in `oscard.conf`.

4.4.2 Oscard Internals

In this section we will describe how Oscard internal works, and how it can be configured. We will also clarify how we use pseudo-randomness within simulations.

[16] `http://bottlepy.org/docs/dev/index.html`

Each randomized decision in Oscard is taken using a randomizer obtained through `oscard.randomizer.get_randomizer`. Each time we get a randomizer, it is initialized with a seed taken from a configuration file[17] or, in case one is not specified, directly from Bifrost's last simulation ID (The seed used is equivalent to the simulation ID that the user wants to execute). It is for this reason that every randomized decision can be repeated simply setting the seed in the configuration file.

Every simulation is composed of a precise number of commands[18] run in sequence. Each command is executed at a precise step which is a discrete instant in time. Currently available commands are *create*, *resize*, *destroy* and *NOP*. The first three commands are clear in their intent; the last one, i.e. the *NOP* command, is a "no operation" command. It is meant to make simulations more realistic in the sense that, in reality, it is impossible that at each time instant the system is asked to perform a CRD[19] operation. *NOP* operation simulates the fact that the system could be idle (in term of requests from users) in some moments. "No operation"s allow us to change operation *density* along time.

At each step, Oscard chooses a command at random and executes it. Commands can have different weights[20] that influence their probability to be chosen. Each command is executed *when and only when* the command before has been completed (either successfully or not); thus, the state of the machine that is interested in the operation, can be one of `ACTIVE` or `ERROR`[21]. Because of this reason, and because Oscard is single-threaded, we can say that Oscard's simulations are run *serially*. This fact is important, because in conjunction with pseudo-randomness, it ensures *simulation repeatability*. Repeatability is ensured for what concerns Oscard itself. It could be, in fact, that the same

[17]In `oscard.conf`: `random_seed`
[18]In `oscard.conf`: `no_t`
[19]Create Resize Destroy
[20]In `oscard.conf`: `<command-name>_w`
[21]Two of possible instance states in OpenStack. See `http://docs.openstack.org/developer/nova/devref/vmstates.html`.

simulation brings to different results not because of Oscard's decisions, but because of OpenStack's internal behavior. *Simulation repeatability* has to be interpreted inside Oscard; thus, commands executed in two simulations with the same seed will be equal and equally executed. However, we cannot state that their execution will bring OpenStack to the very same internal status. It could be, for example, that the same create operation ends successfully or not. This fact has to be attributed to OpenStack and *not* to Oscard itself. OpenStack, in fact, is a very vast system and we are not supposed to control its behavior directly from Oscard. In other words, we can ensure, as *Guidelines for Evaluating and Expressing the Uncertainty of NIST Measurement Results* states, to meet repeatability conditions: "the same experimental tools"; "the same observer"; "the same measuring instrument, used under the same conditions"; "the same location"; "repetition over a short period of time" and the "same objectives". We *cannot* guarantee that the experiment (i.e. the simulation) is repeatable in its strictly scientific meaning[22].

As already said, Oscard is composed of two parts, the server and the client one. Oscard's proxy (the server part) can be run both as a Docker container or running `./bin/run_proxy` from a shell[23]. Oscard, as FakeStack, has a source file called `oscardrc`. Once the user runs `source oscardrc`, `run_oscard` script is available in `PATH`, this script can be used to start Oscard's container.

Oscard is highly configurable, but it is important to note that each option has a default value (see `https://github.com/affear/oscard/blob/master/oscard.sample.conf`). Some options are relevant only for server, others for client and some for both. We list their meaning and split them between the two Oscard components to better understand them.

The server part can be configured in terms of:

- `proxy_port`: sets the port on which Oscard's proxy will listen on.

[22]http://en.wikipedia.org/wiki/Repeatability

[23]Dockerized version is recommended, because it allows the user not to install all Oscard's dependencies before running

- `os_username`, `os_tenant`, `os_password`: access credentials for the user used in OpenStack.
- `fake`: if set to `True`, `FakeAPI` will be used.
- `ctrl_host`: the IP of the docker container running controller node.
- `fb_backend`: it's the Firebase backend URL.
- `random_seed`: the seed that `NovaAPI` will use to choose random instances and random flavors. Set this parameter to the ID of the simulation that needs to be run.

The client part can be configured in terms of:

- `fb_backend`: as above.
- `random_seed`: as above.
- `no_t`: the number of steps for the simulation.
- `create_w`, `resize_w`, `delete_w`, `nop_w`: weights for commands.
- `proxy_hosts`: the URLs for the proxies on which the simulation will be run concurrently (e.g. `host1.example.com:3000,host2.example.com:80`). Oscard, in fact, can run the same simulation concurrently on more than one host (if real-time comparisons are needed).

Figure 4.4.2 shows Oscard functioning. In particular it reports a sequence diagram that illustrates the workflow of a `create` operation.

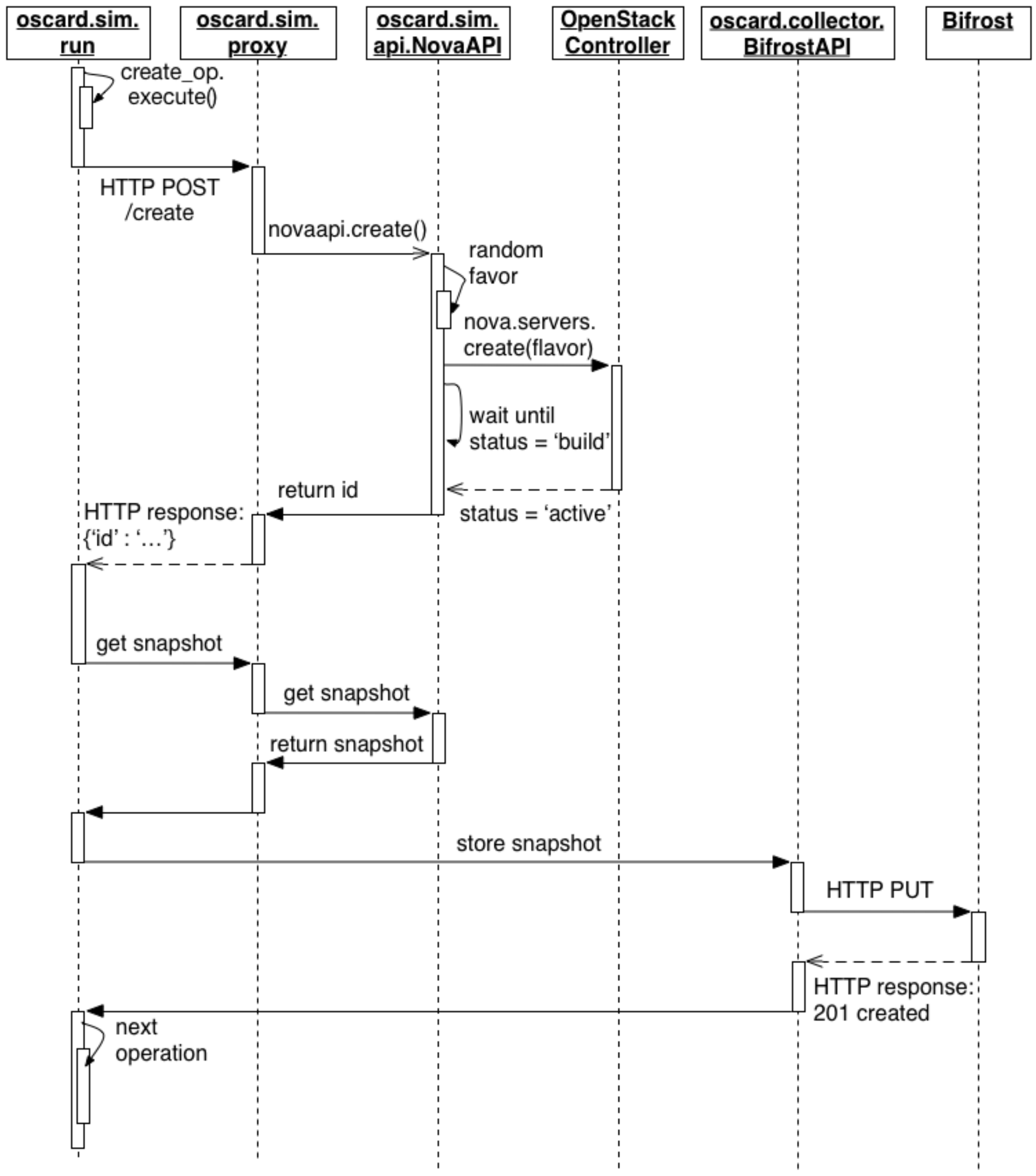


Figure 4.4: Workflow for a `create` operation

4.5 Other Components

The last two components of aDock take care of its backend database and view. These two roles are covered respectively by Bifrost and Polyphemus.

4.5.1 Bifrost

Bifrost[24] is the name for the Firebase application that we use to store simulation data output. Firebase uses a non-relational JSON database. The whole aDock database is thus a JSON structure that is exportable in a `.json` file. Firebase provides a JavaScript and a python SDK and it natively supports real-time notifications on data change (only for JavaScript SDK). We decided to use this backend type because of its SDKs; for the portability of `.json` format; for the advantages of dealing with a non-relational database when data is very mutable (especially while developing); because performance is not needed in our case and because of its reliability being a cloud service. It is important to say that Firebase, by design, offers support for concurrent calls[25]. However, we interact with Firebase's APIs (in Oscard's `BifrostAPI`) using a python library which doesn't support them[26]. This is the reason why our simulations *cannot* run concurrently; i.e. the *same* simulation can run concurrently on more hosts (no problem, Oscard supports it), but *two different* simulations cannot run concurrently (on different hosts).

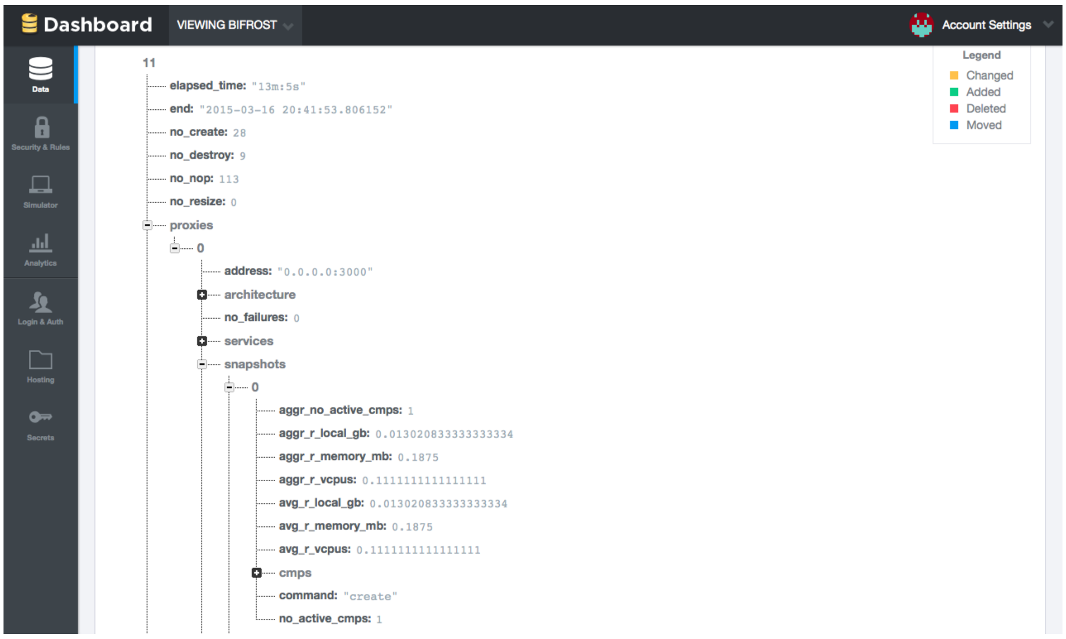
Firebase offers a dashboard (see figure 4.5.1) that is updated in real-time and allows the user to perform CRUD[27] operations on data.

[24]`https://bifrost.firebaseio.com/`

[25]`https://www.firebase.com/docs/web/guide/saving-data.html#section-transactions`

[26]`http://ozgur.github.io/python-firebase/`

[27]Create Read Update Delete



```
11
├── elapsed_time: "13m:5s"
├── end: "2015-03-16 20:41:53.806152"
├── no_create: 28
├── no_destroy: 9
├── no_nop: 113
├── no_resize: 0
├── proxies
│   └── 0
│       ├── address: "0.0.0.0:3000"
│       ├── architecture
│       ├── no_failures: 0
│       ├── services
│       └── snapshots
│           └── 0
│               ├── aggr_no_active_cmps: 1
│               ├── aggr_r_local_gb: 0.013020833333333334
│               ├── aggr_r_memory_mb: 0.1875
│               ├── aggr_r_vcpus: 0.1111111111111111
│               ├── avg_r_local_gb: 0.013020833333333334
│               ├── avg_r_memory_mb: 0.1875
│               ├── avg_r_vcpus: 0.1111111111111111
│               ├── cmps
│               ├── command: "create"
│               └── no_active_cmps: 1
```

Figure 4.5: Firebase Dashboard

4.5.2 Polyphemus

Polyphems[28] is the Polymer[29]-powered view of aDock. It can be run in a Docker container or not[30]. Its aim is to show data to the user in a friendly way. Polyphemus shows each simulation snapshot in terms of overall average resource usage[31] (through line charts) and percentages of resource usage[32] for each compute node (through bar charts). A tool-bar representing data aggregates[33] for each host is always visible on the top. It shows charts for each of the hosts on which the current simulation is running, giving the possibility to intuitively make comparisons. Moreover it includes more information such

[28]`https://github.com/affear/polyphemus`

[29]`https://www.polymer-project.org/0.5/`

[30]Dockerized version is recommended, because it allows the user not to install all Polyphemus' dependencies such as `NodeJS` before running

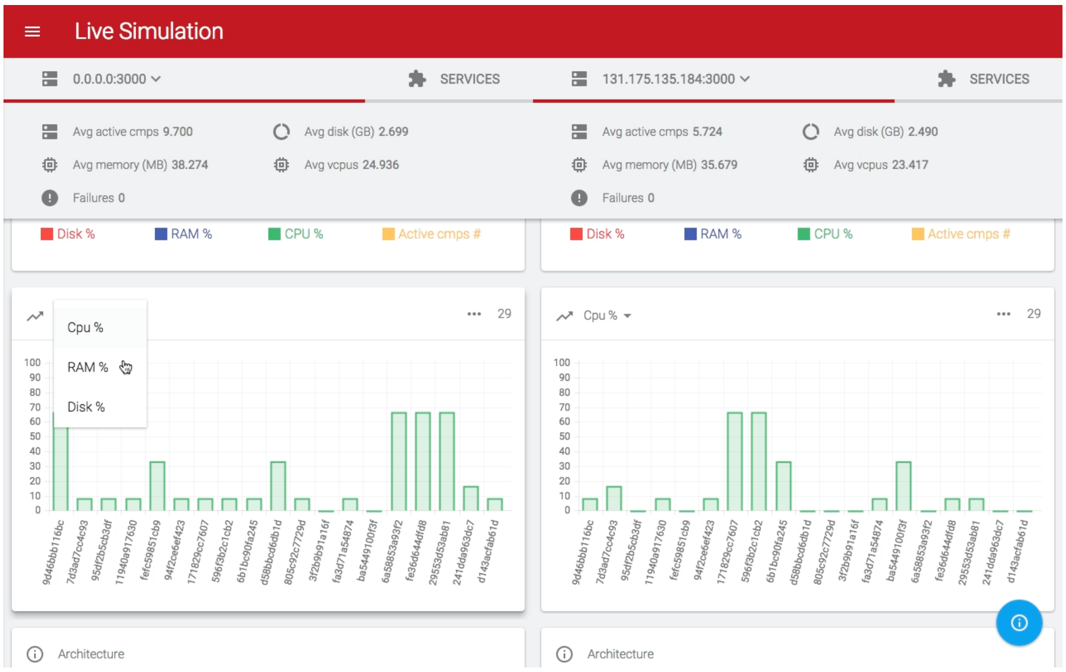
[31]The average of the percentage of vCPUs, memory and disk used calculated on all active compute nodes (nodes that are hosting at least one instance).

[32]The percentage of vCPUs, memory and disk.

[33]The average of all averages of resource usage calculated on the number of simulation steps executed.

as the architecture of each OpenStack system running on each host; active services and simulation progress.

Every information displayed by Polyphemus is updated in *real-time*. In figure 4.5.2, we provide a sample screen shot of Polyphemus.



0.0.0.0:3000

Avg active cmps 9.700
Avg disk (GB) 2.699
Avg memory (MB) 38.274
Avg vcpus 24.936
Failures 0

131.175.135.184:3000

Avg active cmps 5.724
Avg disk (GB) 2.490
Avg memory (MB) 35.679
Avg vcpus 23.417
Failures 0

Figure 4.6: Screenshot of Polyphemus

4.6 aDock's Architecture

aDock is a modular system, where each component is run in its dockerized version[34]. In figure 4.6, a sample aDock architecture is shown.

The simulation is started from a normal laptop using Oscard. The Oscard client contacts Oscard proxies on each of the hosts (specified in `proxy_hosts`) using the exposed endpoints, each endpoint identifies a different command to be executed. For each host, Oscard proxy uses `NovaAPI` to "send" the command to controller node (whose IP is specified in `ctrl_host`). For each host, and at each step, the proxy collects data from the controller using `NovaAPI`

[34]Apart from Bifrost.

methods and stores them into Bifrost using `BifrostAPI`. Data is available and can be consulted connecting to Polyphemus[35] using a web browser on user's laptop.

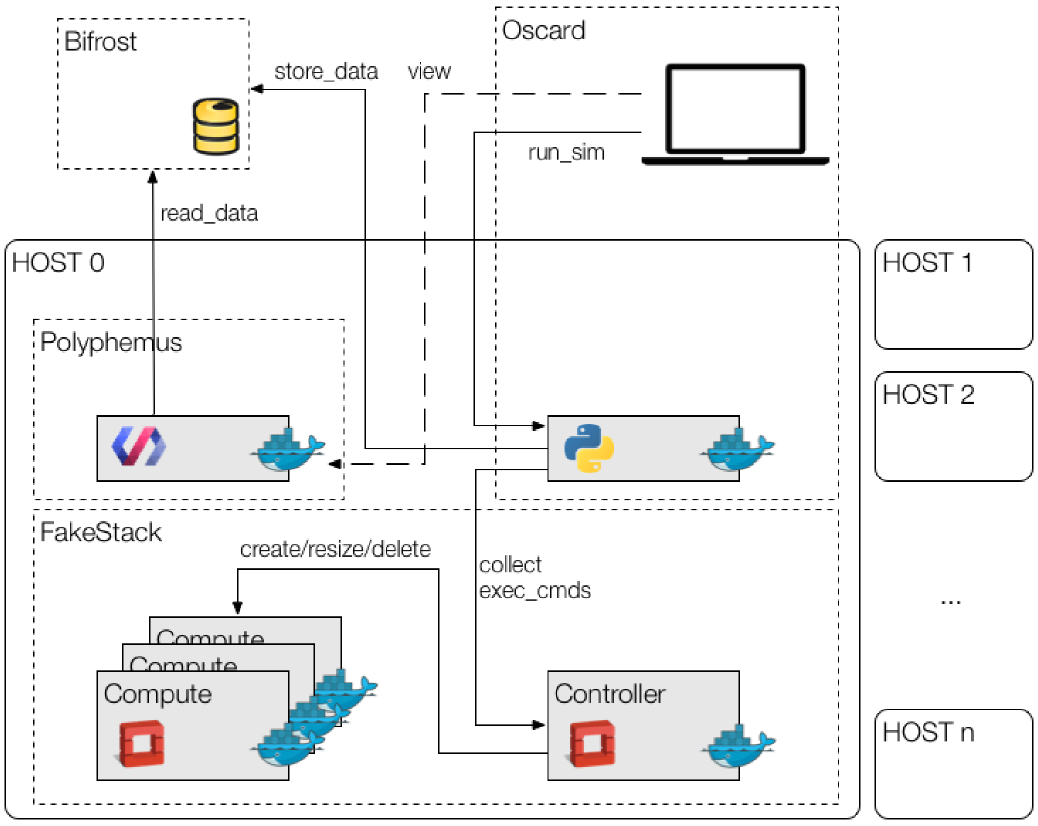


Figure 4.7: aDock architecture

[35]Polyphemus container can be started everywhere, not only on one of the hosts.

Chapter 5

Nova Consolidator

OpenStack already performs virtual machine placement. This is accomplished thanks to its `nova-scheduler` service. Once a virtual machine is created (or, in certain cases, resized or live migrated) the scheduler decides which of the available compute nodes can host[1] the virtual machine (this phase is called *filtering*) and then selects the best[2] among them (this phase is called *weighting*).

OpenStack, on the other hand, *does not* perform virtual machine consolidation. Each of the operations on virtual machines are issued by the user that owns them (or by `Heat` for him/her).

Virtual machine consolidation is a technique by which virtual machines locations on hosts are changed to achieve a better resource utilization in the whole system. Thus, virtual machines are periodically migrated to other hosts if some policy determines that its place is wrong in that precise moment. The policy adopted is determined by the *consolidation algorithm* that is used.

To add virtual machine consolidation to OpenStack we added a service to `Nova` called `nova-consolidator`. The new service is implemented in module `nova.consolidator` which provides a `nova.consolidator.base.BaseConsolidator` class which can be extended

[1]The policies by which a node can host or not a virtual machine are defined by the precise filter which scheduler has been equipped with.

[2]Again, it depends on which weighter is used.

to write custom consolidators (see section 5.1). We also developed some consolidation algorithms, both custom and taken from the state of the art (see section 5.2).

5.1 Consolidator Base

Almost every service in OpenStack has three main components: the *command*[3] (its function `main` will be executed at service startup[4]); the *manager*[5], which contains the service's real logic and the *RPC*[6] *API*[7], which is used by OpenStack services to communicate[8].

The command basically instantiates a `nova.service.Service` object with the name "`nova-consolidator`". The service, in turn, instantiates a `nova.consolidator.manager.ConsolidatorManager` object; and starts its RPC server and its *periodic tasks*. As we can see in listing 5.1[9], `ConsolidatorManager` exposes one periodic task which is represented by the `consolidate` method. Its period is defined in `/etc/nova/nova.conf` file (which can be edited using DevStack. See 4.3.3), under the option `consolidation_interval`. In the configuration file one must also specify the consolidator class to be used by the manager. When the manager is created it creates the specified `consolidator_class` object and periodically calls the method `consolidate`. The `consolidate` method *delegates* consolidation to the consolidator object, from which it obtains the migrations to be performed. Keep in mind that the consolidator could decide not to return any migration in

[3]In our case, `nova.cmd.consolidator`.

[4]When DevStack runs `python setup.py install`, *PyPI* generates an executable file placed at `/usr/local/bin` called `nova-consolidator` (see note 14). It is necessary to make DevStack aware of the new service created to make it install and start it. As a result we had to fork DevStack repository and edit the function `start_nova_rest` in `/lib/nova` (see `https://github.com/affear/devstack/blob/n-cons/lib/nova`).

[5]In our case, `nova.consolidator.manager`.

[6]Remote Procedure Call

[7]In our case, `nova.consolidator.rpcapi`

[8]`https://github.com/affear/nova/tree/n-cons/nova/consolidator`

[9]The code has been properly cut to fit the page and the reader needs.

```python
class ConsolidatorManager(manager.Manager):

  def __init__(self, *args, **kwargs):
    self.compute_api = compute_api.API()
    self.consolidator = importutils.\
      import_class(CONF.consolidator_class)()
    # lines skipped

  @periodic_task.\
    periodic_task(spacing=CONF.consolidation_interval)
  def consolidate(self, ctxt):
    migrations = self.consolidator.consolidate(ctxt)
    for m in migrations:
      self._do_live_migrate(ctxt, m)

  def _do_live_migrate(self, ctxt, migration):
    instance = migration.instance
    host_name = migration.host.host
    # exception catching skipped
    self.compute_api.live_migrate(
      ctxt, instance,
      False, False, host_name
    )
```

Listing 5.1: Code for `nova.consolidator.manager.ConsolidatorManager`

case they would not improve system status. Once the migrations are obtained, they are applied using `nova-compute`'s API.

The consolidator class is, by default, `nova.consolidator.base.BaseConsolidator` (see listing 5.2[9]), which does nothing but define a base class to extend with real consolidation algorithms. Its `get_migrations` method, in fact, returns an empty list of migrations.

`consolidate` method obtains a snapshot of the system (see subsection 5.1.1 for snapshot object details) and passes it to the `get_migrations` method. `get_migrations` will implement the desired consolidation algorithm. Eventually, a transitive closure on migrations is applied[10] and the migrations are returned to the manager.

[10]If instance I is moved first to host A and then to host B; instance I is only moved to host B.

```
class BaseConsolidator(object):

  class Migration(object):
    def __init__(self, instance, host):
      super(BaseConsolidator.Migration, self).__init__()
      self.instance = instance
      self.host = host

  # _transitive_closure method
  # implementation skipped

  def consolidate(self, ctxt):
    snapshot = Snapshot(ctxt)
    migs = self.get_migrations(snapshot)
    return self._transitive_closure(migs)

  def get_migrations(self, snapshot):
    return []
```

Listing 5.2: Code for `nova.consolidator.base.BaseConsolidator`

5.1.1 Objects

We thought that it was not a good idea to ask the user to learn and understand OpenStack's complex database APIs. Due to this fact, we developed `nova.consolidator.objects`, a module that defines the abstraction of system snapshots used in method `get_migrations`. The module provides the class `nova.consolidator.objects.Snapshot`. A `Snapshot` object offers attributes to access all information about the system, such as the currently active nodes, and the currently active instances (also per node). The `Snapshot` is renewed at each consolidation cycle. Attributes are lazily obtained on their first call; subsequent invocations won't refresh snapshot's state. The Snapshot is, thus, entirely cached: when an instance or a compute node is asked and returned, it will not be queried again on OpenStack's database. Its status will always be *frozen* at the moment the first query has been performed. To refresh a `Snapshot` it is necessary to create a new `Snapshot` object.

In detail, a `Snapshot` object offers all instances (`instances` attribute); running instances (`instances_running` attribute); both those instances that are migratable and those that are not[11] (`instances_migrable` and `instances_not_migrable` attributes, respectively) and active nodes (`nodes` attribute). Instances are `nova.objects.instance.Instance`[12] objects; nodes are wrappers for `nova.objects.compute_node.ComputeNode`[13] objects, which add the possibility to get all, running, intances that are migratable and not, per compute node.

In any case, the developer does not instantiate `Snapshot` objects: this is up to `consolidate` method, which already instantiates and passes the current system snapshot to method `get_migrations` . `get_migrations` is therefore the *only* method that needs to be overridden by the user in a custom consolidator class.

In listing 5.3 we provide an example of using a `Snapshot` in a python script.

5.2 Algorithms

In this section, we explain the consolidation algorithms that we implemented in our `nova-consolidator`. Each of the proposed algorithms is run inside a consolidator class that inherits from `nova.consolidator.base.BaseConsolidator` .

5.2.1 Random Algorithm

The first algorithm we implemented is a random one[14]. This algorithm was implemented for testing purposes and to see if randomization could bring im-

[11]According to us, an instance is *migratable* when its state is `ACTIVE` and its power state is `RUNNING`.

[12]`https://github.com/openstack/nova/blob/master/nova/objects/instance.py`

[13]`https://github.com/openstack/nova/blob/master/nova/objects/compute_node.py`

[14]`https://github.com/affear/nova/blob/n-cons/nova/consolidator/base.py`

```
from nova import config, objects, context
from nova.consolidator.objects import Snapshot

# Init operations
config.parse_args('')
objects.register_all()
ctxt = context.get_admin_context()

# Using the Snapshot
s = Snapshot(ctxt)
nodes = snapshot.nodes # all compute nodes
node = nodes[0] # the first node
instances = node.instances # all instances on that node (list)
print node.vcpu
print node.id
print instances[0].flavor
# `node` has all attributes as
# nova.objects.compute_node.ComputeNode has,
# as well as `instances[0]` has all attributes as
# nova.objects.instance.Instance has.

nodes_new = snapshot.nodes
# nodes are not refreshed because they are cached!
assert nodes == nodes_new # evaluates to True
```

Listing 5.3: An example of using a `Snapshot` object

provement in resource optimization, given that virtual machines are never moved in OpenStack[15].

The algorithm needs to be configured with a percentage of instances to migrate to other compute nodes. Instances are randomly chosen from hosts' migratable instances and their destinations are randomly chosen among remaining hosts. Choices dp not take into account host suitability. The algorithm doesn't rely on the fact that migrations will be applied. If a migration fails, due to resource usage problems, it is not a problem.

The random algorithm is highlighted in in listing 5.4[9].

[15]Except for when a user decides to, or on a resize call. When a virtual machine is resized to a flavor which is too big for the current host, it is migrated to a suitable one.

```
def get_migrations(self, snapshot):
  nodes = snapshot.nodes
  no_nodes = len(nodes)
  migration_percentage = float(CONF.consolidator.migration_percentage) / 100
  no_inst = len(snapshot.instances_migrable)
  no_inst_migrate = int(no_inst * migration_percentage)

  # if no_inst_migrate == 0
  # or no_nodes < 2, then
  # return empty list.
  # Cannot migrate.

  migs = []
  while no_inst_migrate > 0:
    nodes_cpy = list(nodes) # copy nodes list

    from_host = choose_host(nodes_cpy)
    # choose_host code is skipped.
    # The chosen node is randomly chosen
    # taking into account that it has to host
    # at least one instance.

    inst_on_host = from_host.instances_migrable
    no_inst_on_host = len(inst_on_host)

    top_bound = min(no_inst_on_host, no_inst_migrate)
    n = random.randint(1, top_bound)
    no_inst_migrate -= n

    instances = random.sample(inst_on_host, n)
    nodes_cpy.remove(from_host) # do not choose same host
    to_host = random.choice(nodes_cpy)
    for i in instances:
      migs.append(self.Migration(i, to_host))

  return migs
```

Listing 5.4: Code for random algorithm

5.2.2 Genetic Algorithm

The idea to use a genetic algorithm to solve virtual machine consolidation problem is taken from the state of the art (see section 3.2.1), although we heavily revisited it[16].

Our genetic algorithm uses a list as chromosome structure. Each element of the list (a gene) is considered to be a migratable instance, and its value is the hostname of the compute node that will host the instance. At first, we developed the algorithm as a "standard" genetic algorithm. So, it provided a crossover step. A child generated by crossover is considered *unhealthy* when it violates system constraints (instances on a node exceed its vCPU, memory or disk capacity). After some simulations we realized that 100% of the children generated were unhealthy. Suddenly, we realized that the probability of generating a healthy child was close to zero because of the tightness of system constraints. Thus, the crossover step became useless and we decided to turn it into a massive mutation. In the crossover we would chose[17] two chromosomes, the father and the mother, and cross them[18]; now we only choose one chromosome and massively[19] mutate it.

The algorithm is configurable in all of its aspects:

prob_mutation (Defaults to 0.8) The probability to apply mutation on a chromosome.

mutation_perc (Defaults to 10) The percentage of genes to be mutated in a chromosome, once mutation is decided to be applied.

selection_algorithm

(Defaults to `nova.consolidator.ga.functions.RouletteSelection`)

The selection algorithm used. The selection algorithm plays its

[16]https://github.com/affear/nova/tree/n-cons/nova/consolidator/ga

[17]Chromosome are chosen among the whole population using a specific selection algorithm.

[18]We performed a single point crossover. See `http://en.wikipedia.org/wiki/Crossover_%28genetic_algorithm%29#One-point_crossover`.

[19]We change the value of a high percentage of its genes.

role when it's time to decide which chromosomes to cross (in our case, mutate) to generate a new child to add to the population (an implementation of tournament selection is provided in `nova.consolidator.ga.functions.TournamentSelection`).

fitness_function
(Defaults to `nova.consolidator.ga.functions.NoNodesFitnessFunction`) The Fitness function establishes how much the chromosome fits the desired solution (see listing 5.5 for `NoNodesFitnessFunction` implementation).

population_size (Defaults to 500) The size of the population.

epoch_limit (Defaults to 100) The number of epochs after which the algorithm stops.

elitism_perc (Defaults to 0) The percentage of chromosomes that will pass to the next epoch. The number `N` of elite chromosomes is determined from this option and the `population_size`. At each step the best `N` chromosomes (according to the fitness function used) will pass to the next epoch.

There is another option which is `best` (defaults to *False*). After running some simulations, we discovered that most of the epochs run without improving the fitness of the best chromosome, meaning we spent a lot of time generating useless children. To overcome this problem we revisited the mutation. When we apply mutation we change a gene's value and maintain the chromosomes' validities. Changing a gene's value means moving an instance to another compute node. The other compute node, normally, is chosen randomly among suitable nodes[20]. When `best` is set to *True*, the other compute node is no longer chosen randomly; instead we choose the best node[21] among the suitable

[20]Nodes that, hosting the machine, will not exceed their capacity in terms of vCPUs, memory and disk.
[21]The most busy compute node.

```python
class NoNodesFitnessFunction(FitnessFunction):
  # The higher the less nodes are used:
  # - no_nodes = 1: fitness = 1
  # - no_nodes -> infinite: fitness -> 0

  def get(self, chromosome):
    return float(1) / len(set(chromosome))
```

Listing 5.5: Code for `NoNodesFitnessFunction`

compute nodes. With this change in mutation logic, it turns out that the best chromosome generated in the very first epoch will almost never be exceeded by another one. Thus, this variant, truncates to number of epochs to 1. The "best" variant is something vaguely similar to a genetic algorithm because there is no evolution except from the selection logic and the mutation.

In algorithm 3 we provide a high-level pseudo-code for our genetic algorithm.

Algorithm 3 Pseudo-code for our genetic algorithm

```
population = population_size random generated valid chromosomes
epoch_count = 0

procedure NEW_CHROMOSOME                                  ▷ Returns a new chromosome
    Select a chromosome from population using selection_algorithm
    Mutate the chromosome with probability mutation_prob
    Return the chromosome obtained
end procedure

procedure NEXT                                            ▷ Returns next population
    Take the elite from current population (elitism_perc)
    Add it to new population
    while new population is not as big as population_size do
        Add to new population the result of new_chromosome procedure
    end while
end procedure

while epoch_count is less than epoch_limit do
    population = next()
    Increment epoch_count
end while

return population
```

5.2.3 Holistic Algorithm

We also provide a holistic algorithm[22] taken from the state of the art (see section 3.2.2).

The algorithm takes the least loaded compute node and tries to move all its instances to the most loaded node that can host them. The algorithm tries to move instances from the biggest to the smallest (in terms of resource usage). When finished with the least loaded node, the algorithm examines the second least loaded node and so on, until all nodes are examined.

In algorithm 4 we provide the pseudo-code for the holistic algorithm.

Algorithm 4 Pseudo-code for holistic algorithm

```
nodes = nodes from given snapshot
no_nodes = number of nodes given in snapshot
new_state = mappings (instance: node)

for all node in nodes do
    node = least loaded node

    if node has no instances then
        continue
    end if

    Sort node's migratable instances from biggest to smallest

    for all instance in node's instances do
        to_node = most loaded node that can host instance
        if to_node doesn't exist then
            continue
        end if
        add mapping (instance: to_node) to new_state
    end for
end for

return new_state
```

[22] `https://github.com/affear/nova/tree/n-cons/nova/consolidator/holistic`

Chapter 6

Evaluation

Due to the two-topic nature of this thesis we split this chapter into two sections. Section 6.1 is about aDock system evaluation, while section 6.2 is about the evaluation of the different consolidation algorithms implemented in OpenStack.

6.1 aDock

This section presents the results of the experiments we carried out to evaluate aDock's capability to create fully functional experimentation environments based on OpenStack, and its scalability.

The first experiments we show were performed on a Dell PowerEdge T320 server[1]. This is not a high-end server, and can be bought nowadays for less then one thousand euros.

In the experiment we created an aDock environment with 1 controller container and 1 compute container. We then progressively increased the number of compute containers to identify how many could be run at the same time. Keep in mind that each container was actively running OpenStack code. The maximum number of compute nodes that can be run in a two-node architecture with legacy networking, before the controller becomes a management bottle-

[1]Intel Xeon E5-2430 2.20GHz, 15M Cache, Ubuntu 14.04LTS 3.13.0-32-generic X86_64. 16GB of RAM and SWAP. No SSD equipped.

neck, is 20 [2]. Therefore, we wanted to see whether we could reach this threshold on a single machine, and to what extent we could surpass it. Table 6.1 shows the results of our experiments.

Config	**AvgTime** [sec]	**AvgCPU** [%]	**AvgMem** [%]	**AvgSwap** [%]
clean	588	0.16	1.875	0
1 + 0	188	2.295	30.956	0
1 + 1	185	2.707	38.076	0
1 + 6	182	5.616	65.979	0
1 + 12	189	5.478	98.847	0.038
1 + 22	191	5.54	98.869	0.257
1 + 42	214	7.59	98.978	21.865

Table 6.1: aDock's performance on a PowerEdge T320 server.

As we can see we succeeded in reaching "1 + 20" architecture and overcome it to "1 + 42". We think this is a great result, because it could possibly allow the user to try different architectures with less compute nodes and more controller nodes. Although, up to now, aDock doesn't support architectures with more than one controller node by default.

On of our aims is to understand if a user can use aDock on his/her laptop without owning a server. So, we tried to deploy an aDock environment on two different laptops. We left Google Chrome[3] (our favorite web browser) and Sublime Text[4] (our favorite text editor) running, because we assumed that a user is developing and browsing while using aDock platform[5].

Our goal was to deploy a "1 + 5" configuration (one controller node and five compute nodes), which we think it is a configuration which satisfies most of testing use cases. The test took place with the same form of the server one, except from the fact that we stopped at "1 + 5" architecture goal. In table 6.2 we show the results of the experiment conducted on a Samsung SERIES 5

[2]`https://docs.chef.io/openstack_architecture.html#openstack-chef-single-controller-n-compute`

[3]`https://www.google.it/chrome/browser/desktop/`

[4]`http://www.sublimetext.com/`

[5]Keep in mind that this fact impacts considerably the test. Google Chrome, for example, increases resource usage so much, that Google itself provides ways to lower it (see `https://support.google.com/chrome/answer/6152583?hl=en`).

ULTRA[6], while in table 6.3 we show results on an Apple MacBook Pro (Early 2011)[7].

Config	AvgTime [sec]	AvgCPU [%]	AvgMem [%]	AvgSwap [%]
clean	1736	12.34	52.052	7.779
1 + 0	898	12.495	95.954	9.809
1 + 1	923	12.77	96.909	19.235
1 + 2	934	13.14	96.528	29.861
1 + 3	976	13.52	96.048	38.053
1 + 4	1104	13.79	96.453	43.665
1 + 5	—	14.02	96.325	51.496

Table 6.2: aDock's performance on a Samsung SERIES 5 ULTRA.

Config	AvgTime [sec]	AvgCPU [%]	AvgMem [%]	AvgSwap [MB]
clean	466	3.05	93.63	55.5
1 + 0	242	9.76	99.38	93.8
1 + 1	255	12.78	99.75	93.8
1 + 2	255	14.94	99.75	93.8
1 + 3	257	15.91	99.75	93.8
1 + 4	288	16.79	99.75	93.8
1 + 5	—	18.01	99.75	93.8

Table 6.3: aDock's performance on a Apple MacBook Pro (Early 2011).

We succeeded in deploying a "1 + 5" configuration on both laptops, maintaining a usable environment. With the term "usable", we mean that the user can still work on his/her text editor, web browser and aDock itself, and so he/she can go on developing, browsing and run simulations with Oscard with a reasonable response time from his/her laptop. For each step we recorded CPU usage, RAM usage, SWAP usage and the required time to run the next aDock node in that state (*AvgSwap* is expressed in MB for MacBook Pro, because Mac Os dynamically allocates SWAP space and, so, it is not possible to give a percentage of usage.).

[6]Intel Core i5 1.6 GHz, Linux Mint 3.13.0-24-generic XFCE, 4GB of RAM and SWAP. No SSD equipped.

[7]Intel Core i5 2.3 GHz, Mac Os X Yosemite, 8GB of RAM, SWAP is dynamically allocated. SSD equipped.

In the case of Samsung, we can see that there is little dependence among CPU usage, startup time and number of containers. RAM usage and SWAP are strictly correlated, instead. Once RAM usage reaches around 96 percent, SWAP memory starts to be used, resulting in growing percentages of SWAP usage. Thanks to this data, we understand that running a containers is mostly a memory intensive task.

In the case of MacBook, we see CPU usage grow significantly and RAM and SWAP stay almost unchanged during all the steps of the test. Our opinion is that Mac OS is too opaque to the user to understand what is happening to the memory.

It is not surprising to see that MacBook is almost 4 times faster than Samsung and very close to PowerEdge T320 in starting nodes. The MacBook, in fact, is equipped with an SSD hard-drive and Docker stores containers and the images they come from to disk. Moreover SWAP memory is allocated on the disk itself and, when aDock comes to use that, SSD makes the difference.

If we sum up boot times for the "1 + 5" configuration we obtain around 26 minutes for PowerEdge T320[8]; around 1 hour and 50 minutes for Samsung[9] and around 30 minutes for MacBook Pro[10]. We think these are reasonable timings to deploy a private cloud system. We have to keep in mind that Samsung, which resulted in a very high time of deploy, is a laptop which is not to be considered as a default in these years. Its specifics, in fact, are beneath the ones of normal laptops in sales into stores now.

Another important fact to keep in mind is that OpenStack installation through DevStack is a network intensive task due to OpenStack's repositories cloning. All test were run with a connection of 100Mb/s download speed. Timings reported are dilated by the fact that compute nodes are started serially. If they were started concurrently (as FakeStack gives the opportunity to do. See sub-section 4.3.2.) timings would have been lower. Timings considered are

[8]Formula used: $(588s + 188s * 5)/60s$.

[9]Formula used $(1736s + 898s + 923s + 934s + 976s + 1104s)/3600$.

[10]Formula used: $(466s + 242s + 255s + 255s + 257s + 288s)/60s$

to be thought of as worst case scenarios.

6.2 Consolidators

This section presents the results of the experiments we carried out to evaluate the goodness of the consolidation algorithm proposed in section 5.2.

We run the *same* 50 simulations[11] for each of the different consolidators on a "1 + 10" architecture deployed on a Dell PowerEdge T320 server (see 6.1, for server's specifications.). Each simulation was composed of 150 steps (`no_t`=150) and started with an empty system (no running instance). Each of the 10 compute nodes was equipped with 18 vCPUs, 24576 MB of RAM and 3072 GB of disk. Each simulation was configured with a *NOP* operation weight of 20 (`nop_w`=20); create operation weight of 4 (`create_w`=4); destroy operation weight of 1 (`delete_w`=1) and resize operation weight of 0 (`resize_w`=0)[12]. Every consolidator was configured with a consolidation interval of 10 seconds (`consolidation_interval`=10), which we think is unfeasible in a real cloud system. However, we set it according to the time that Nova's `FakeDriver` requires us to create and destroy an instance. This time is much lower than the time that would take `LibvirtDriver` to accomplish the same operation. `FakeDriver`, in fact, only has to create an object and store it in the database. `LibvirtDriver`, instead, spawns a real virtual machine. We chose the specified consolidation interval in order to to make consolidators heavily influence simulation results: a simulation of 150 steps takes about 13 minutes to run, thus executing an operation approximately every 5 seconds; so, we have that the consolidator takes decisions about instance location approximately every

[11] We used the same 50 different seeds in Oscard for each group of simulations (for an explanation of the role of random seeds in Oscard, see sub-section 4.4.2).

[12] We had to remove resize operations from the simulations due to a known bug (see `https://bugs.launchpad.net/nova/+bug/1430057`) which involves Nova's `FakeDriver`, live-migration and resize operation. The bug is tagged as "invalid" because "[. . .] This is just beyond scope of the current fake driver [. . .]". We think that the lack of resize operations doesn't compromise simulation results. It's create and destroy operations that are the real building blocks of a cloud system.

2 operations executed on the system; in this way, consolidators act as soon as possible to “repair” the system, highly influencing its status.

We report here configurations for each of the consolidator used (for a reference of the options see 5.2).

Vanilla No configuration required.

Random `migration_percentage`=20

Genetic Algorithm

- `prob_mutation`=0.8
- `mutation_perc`=10
- `selection_algorithm`=
  `nova.consolidator.ga.functions.RouletteSelection`
- `fitness_function`=
  `nova.consolidator.ga.functions.NoNodesFitnessFunction`
- `elitism_perc`=20
- `population_size`=500
- `epoch_limit`=100

Genetic Algorithm (“best” variant)

- `prob_mutation`=0.8
- `mutation_perc`=10
- `best`=*True*

Holistic Algorithm No configuration required.

Results are shown in table 6.4[13].

[13]Results are truncated at the third decimal digit.

Cons	vCPUs [%]	RAM [%]	Disk [%]	BusyCmps	BusyCmpsSD	DsTime [%]
vanilla	22.918	34.638	2.505	7.753	2.905	0
random	26.861	40.247	2.937	6.617	2.499	8.306
ga	31.413	46.759	3.440	5.367	2.560	9.186
ga_best	37.217	54.638	4.038	4.864	2.370	10.573
holistic	30.598	45.811	3.371	6.143	2.356	8.826

Table 6.4: Results of 50 simulations run on each type of consolidator.

The first column is for the consolidator used; the second, third and fourth column for the percentage of vCPUs, RAM and disk used respectively[14]; the fifth column is for the number of compute nodes active (out of 10); the sixth for the standard deviation of the active nodes and, eventually, the seventh for maximum downscale time[15].

First five columns are clear in their intent, while we explain the role of the last two ones.

BusyCmpsSD is the standard deviation of the number of active compute nodes. We report it to compare how stable consolidators are in maintaining the configuration obtained. Keep in mind that it makes sense only to compare this results among consolidators given that they were subject to the same simulations; the number by itself doesn't say anything about the consolidator itself, because the number of active compute nodes oscillates due to create operations too.

DsTime is the maximum downscale time. We calculate it examining each step of a simulation. If the number of active compute nodes decreases from a step to another, then a downscale window is started. The window is considered closed once the number of active compute nodes increases. The window is not activated if the decreasing is caused by a destroy operation. Given that destroy operation effect id discarded, downscale can only be caused by the

[14]Ratios are calculated only on nodes that have a vCPUs usage greater than 0 (almost the same as saying, "nodes that host at least one instance").

[15]All of the values shown are an average on the 50 simulations of the interested consolidator.

consolidator's effect. *DsTime* is the average on all simulation of the *maximum* downscale window detected in ratio with the total number of steps (in our case, 150). This means that, if we obtain a *DsTime* of 10, the considered consolidator succeeded in getting a downscale for, at maximum, the 10 percent of the steps in a simulation, and so, "15" steps. "Vanilla" configuration, obviously, gets a *DsTime* of 0.

With the weights on operations described above we obtained a mean number of create operations of 26.8; 5.84 destroy operation and 117.36 *NOP* operations.

All of the consolidators brought to an improvement in all metrics examined compared to "vanilla" configuration. The number of active compute nodes decreased, while resource usage increased significantly. The standard deviation of the number of compute nodes decreased, meaning that consolidators succeed in making the system more stable. Maximum downscale time, as already said, increased considerably.

If we compare consolidators among them, then "ga_best" configuration is the one which gives best results on almost all metrics. It is a bit surprising to discover that it gives much better results than standard "ga" configuration. Standard "ga", in fact, is based on evolution as a standard genetic algorithm suggests. "Best" variant is *not* a genetic algorithm indeed. The core of the algorithm itself is only based on the randomness of the generation of chromosomes and on influencing mutation turning it into a non-random one. Chromosomes to be mutated are chosen according to roulette selection and genes to be mutated are chosen as a random sample of chromosome's genes, both in the standard algorithm and in "best" variant. It is surprising that, *discarding evolution in its entirety*, it is enough to move an instance chosen at random to the busiest feasible node (see subsection 5.2.2), instead that to a random one to bring to such a high improvement (about 5 percent on vCPUs usage; about 8 percent on RAM usage and about 1 node active less). Another surprising fact is the comparison between "best" variant and holistic algorithm. While the first seems to act almost randomly, the second seems to perform very sensed actions and calculations (see subsection 5.2.3); however, the holistic algorithm is very far from the improvement given by "best" variant of the genetic algorithm (it is even worse than standard genetic algorithm). It is surprising that even random algorithm brings to such a big improvement with respect to "vanilla" configuration (about 4 percent on vCPUs usage; about 6 percent on RAM usage and about 1 node active less). It could be that "vanilla" OpenStack, with all its default configurations, is so bad at virtual machine placement that there is no way to make the situation worse. In OpenStack, by default, once an instance has to be placed (on creation, for example) the service `nova-scheduler` is invoked. The scheduler returns a list of nodes that can host the instance based on policies which can be configured and customized by the administrator of the system. Given that the scheduler returns a list of possible hosts, a node has to be chosen. The host is chosen according to its *weight*. Weighers are configurable and customizable in turn, but, by default, their behavior is to prefer spread against stacking[16]. This behavior is totally in contrast with virtual machine consolidation.

Another consideration about standard genetic algorithm is a non-functional one: algorithm performance. Genetic algorithms have always to deal with performance problems. This is especially the case given that our code is written in python. Genetic algorithm was implemented avoiding object oriented programming and preferring built-in data structures such as lists, dictionaries and tuples; preferring built-in functions (such as `map`, `reduce`, `filter` and `zip`) and list, dictionary and tuple comprehensions to `for` loops. Even if this decisions give a speed-up to genetic algorithm performance, the algorithm is slow, with an average of 5 seconds run even when it is the case of about 20 instances in the system (the average of instances during each simulation). The computational time could explode in case of hundreds of instances in the system. This fact has to be kept in mind by the system administrator

[16]`http://docs.openstack.org/developer/nova/devref/filter_scheduler.html`

when using this consolidator. “Best” variant is not effected so much by this problem because of its limiting in epoch run (1 instead of 100 by standard configuration).

“Best” variant of genetic algorithm is the best at standard metrics (vC-PUs, RAM and disk usage and number of active compute nodes) and the one that guarantees the highest maximum downscale time (about 10 percent), but holistic algorithm is the most stable one (lowest number of active compute nodes standard deviation) even if it is very very close to “best” variant (a difference of 0.014 percent).

Chapter 7

Conclusions and Future Work

Due to the two-topic nature of this thesis we split this chapter into two sections. Section 7.1 contains conclusions regarding the results obtained while testing the aDock system (see section 6.1), and possible future work. Section 7.2 discusses conclusions regarding the results obtained from testing the consolidation algorithms (see section 6.2), as well as future work in the field of Virtual Machine consolidation in OpenStack.

7.1 aDock

Tests conducted on our system confirmed our suppositions: it is possible for a developer or researcher to develop OpenStack code on his/her laptop and use aDock to run simulations against a fully-functional OpenStack system. The results obtained show reasonable starting times for the entire architecture. We can conclude that aDock is "lightweight". However we think that a comparison between aDock and one of the other options available in the state of the art (e.g. *Chef*. See subsection 3.3.2) should be done. Direct comparison is necessary to understand if aDock is really better then its "competitors" in terms of startup times. Keep in mind that, by construction, Docker containers make aDock more lightweight than an architecture that uses a hypervisor and virtual machines (see paragraph 4.2.3).

Future work on aDock will be vast. First of all FakeStack should become a completely *modular* system. Docker developers strongly advocate small, lightweight containers where each container has a single responsibility. This is not the case in FakeStack, which gives a lot of responsibilities to a single container. FakeStack nodes are “fat containers” that run a lot of different processes. The controller node, for example, in its minimal configuration, runs `rabbitmq-server`; `mysql`; `keystone`; `glance-api`; `glance-registry`; `nova-api`; `nova-cert`; `nova-conductor` and `nova-scheduler`. This is in contrast with Docker’s philosophy and makes FakeStack less “flexible” than it could be. The solution to this problem would be to make each OpenStack service run in a separate container[1]. This change would make FakeStack much more flexible and configurable by the user. To do this we envision providing a templating language, one that would allow FakeStack to automatically deploy an OpenStack architecture as described by a user. This is something that *Chef* and *Puppet* already do (see subsections 3.3.2 and 3.3.3). With the adequate support from the OpenStack community, FakeStack could become the equivalent, but Docker-powered, of *Chef-OpenStack* and *Puppet-OpenStack* in the OpenStack world.

Docker recently released tools for container orchestration[2]. Among these we have *Compose*[3], “a way of defining and running multi-container distributed applications with Docker”. We think that this functionality fits perfectly with what we envision for a more modular FakeStack. Compose allows the user to create a `docker-compose.yml` file and start its newly defined system running `docker-compose up`; Compose will start and run the entire system determining the right order to start containers.

If we want to start a controller node, we could start it using Compose. Listing 7.1 shows a possible `docker-compose.yml` file for a controller node as a proof of concept[4]. The Keystone (`key`) container depends on the MySQL (`db`)

[1]There is already an attempt to this, `https://hub.docker.com/u/cosmicq/`.
[2]`http://blog.docker.com/2015/02/orchestrating-docker-with-machine-swarm-and-compose/`
[3]`http://docs.docker.com/compose/`
[4]Not all necessary services are listed. Ports are avoided. Images are supposed to be

```
key:
  image: fs-key
  links:
   - db
   - rabbit
  ports:
   - ...
  volumes:
   - ./keystone.conf

g-api:
  image: fs-g-api
  links:
   - key
  ports:
   - ...
  volumes:
   - ./glance.conf

n-api:
  image: fs-n-api
  links:
   - key
  ports:
   - ...
  volumes:
   - ./nova.conf

db:
  image: mysql

rabbit:
  image: rabbit
```

Listing 7.1: Sample controller's `docker-compose.yml`

and the RabbitMQ (`rabbit`) containers; while Glance API (`g-api`) and Nova API (`n-api`) containers depend on `key`. All configuration files are specified as volumes, to make it unnecessary to rebuild images if a modification in the configuration happens.

We could also provide the user with built-in *composed* system, such as "all-in-one" and "1 + N" architectures. We could also provide systems for single OpenStack modules. "Nova" composition, for example, could include containers for all of the Nova's services, such as the scheduler, the API and so

available.

on.

In this modular case, every image will map to a single OpenStack service. The images we provide should fit the users' needs, and allow one to specify what OpenStack code to run (as already said in non-functional requirement 1. See 4.2.3). This is why we think that we could still use DevStack to install the single services. This choice would allow the user to choose the GitHub repository URL and branch, and to configure the service itself (see subsection 4.3.3). Regarding configuration, the user should still provide a main `local.conf` as for DevStack; but we could provide a script which parses this file and generates a different configuration file for each of the services configured. The output files (e.g. `keystone.conf` and `nova.conf`) would contain global configurations for DevStack itself, the specific service's configuration (including its repository URL and branch), and the `ENABLED_SERVICES`. This option would be set by the script to the particular service which will be run in the specific container. The script considered should be run before container starting to obtain the different configuration files.

OpenStack configuration is not "hot-reloaded" at every modification, but requires container reboot. We could avoid rebooting (rebooting is heavier then service restarting, which would imply a new DevStack installation) providing scripts to restart services inside containers. This fact is not trivial, because services could have dependencies among them and restarting could break service startup and other related services.

Regarding Oscard, we would like to allow the user to run simulations with more operations (e.g. live-migration). Currently we only support create, resize, destroy and *NOP* operations.

Oscard could run more realistic simulations. We could get data from different real cloud systems to understand how many operations are performed per second, their proportions (e.g. create vs destroy operations), and their density through time. Up until now, in fact, we only supposed that the operation density is not homogeneous (introducing *NOP* operations) and we applied arbitrary proportions between operations.

We would also like to support aggregates of simulations. We would like to be able to run a group of simulations and extract averages of the aggregates that are stored in Bifrost. This was what we needed when we wanted to run groups of 50 simulations, each block with a different consolidation algorithm. To calculate the numbers in table 6.4 (see section 6.2), we wrote a python script that extracted averages of aggregates from each group of simulations, knowing the starting and ending simulation ID of each group. We suppose that such a situation could happen often to users. Oscard should allow the user to give an unique label to a group of simulations and automatically extract the averages and standard deviations of the aggregates that Oscard already calculates. Polyphemus would also need to be updated to display the new data and somehow represent the concept of *groups* of simulations.

7.2 Virtual Machine Consolidation in OpenStack

Tests run on different consolidators confirmed our thoughts regarding OpenStack's Virtual Machine consolidation. Consolidation brings high levels of improvement for resource usage, with respect to "vanilla" OpenStack. The best algorithm we found, in fact, brought a 14% increase in vCPUs usage, a 20% increase in RAM usage, a 1.5% increase in disk usage, and a 30% decrease of active nodes (on 10 total nodes), with a maximum downscale time of about 10%.

In the future, we will extract standard power on and power off timings of different servers from the state of the art and compare them with maximum downscale times obtained. Maximum downscale time, in fact, is intended to be compared with those timings, given that a node, when inactive, can be powered off. If the time in which the node is inactive is too short, it could be a useless turning it off, because the system could need it while this is happening.

It could be that some algorithms do not allow one to power off servers, and so, they give a "fake" improvement. Resource usage increases, but the system cannot exploit this efficiency trait.

We also believe that more efficient consolidation algorithms could be implemented in the future. The state of the art gives a lot of hints about this (see section 3.2.3 and section 3.2.4).

During simulations instances where supposed to run at a maximum workload. It could be interesting to simulate different workloads on instances based on the type of application they are supposed to run. We could extend Nova's `fake` module to comprise a `DynamicWLInstance` which simulates different workloads given an application type. Workload simulation should be developed starting from data taken from the state of the art.

Up until now our metrics have not involved the number of migrations performed. Every live migration performed, in fact, has a cost in terms of energy and time. In the future we will track the number of live migrations performed. It could be that different algorithms bring to different numbers of live migration and, thus, are preferable to others.

During the whole development we clashed with the current development of `nova.virt.fake.FakeDriver`. It seems that `fake` module of Nova isn't evolving, as are other parts of OpenStack project. The *Kilo* version of OpenStack, in fact, is to be released soon and the community is highly focused on it. However, we understood and used the power of the `fake` module, and we want to fix its bugs and enhance it. During `nova-consolidator` service development we also had to fix a bug in the live migration feature[5], as well as to extend it to accept resize operations (see section 6.2). We also had to implement `nova.virt.fake.MStandardFakeDriver` which allows the developer to start a compute node with multiples and sub-multiples of a `nova.virt.fake.StandardFakeDriver` (12 vCPUs, 16384 MB of RAM and 2048 GB of disk) by means of configurations files (`fake_driver_multiplier`

[5]`https://bugs.launchpad.net/nova/+bug/1426433`

option)[6]. This was necessary for us to simulate different architectures and limit spawning instances. By default, in fact, `FakeDriver` offers a standard implementation with 1000 vCPUs, which is too big to reach system saturation (or reasonable usage percentages) in short simulations and a `SmallFakeDriver` (1 vCPU), which is too small to host more than one instance[7]. In the near future we will open blueprints[8] for both `MStandardFakeDriver` and for `DynamicWLInstance`, to improve Nova's "fake" implementation[9]. It could be interesting to somehow simulate the nodes' energy consumption in `FakeDriver`. For now, it would be a nonsense to extract the nodes' energy consumption, given that our nodes are virtualized Docker containers.

In section 6.1 we already said that OpenStack, by default, prefers virtual machine spreading over stacking. It would be interesting to set `ram_weight_multiplier` to a negative value, to make weighers prefer stacking over spreading[10]. In this way OpenStack would be much better at virtual machine placement in a perspective of consolidation. We could start simulations with a fixed value of instances, e.g. 30, and only perform destroy and *NOP* operations, and compare different consolidators in this perspective. It would be useful to see consolidators in action starting from an empty system, and experiment their effect on create operations; however, if placement is performed with a consolidation perspective, it is when instances are deleted that consolidation makes the difference filling the "holes" that are left behind when instances are deleted.

[6]https://github.com/affear/nova/blob/n-cons/nova/virt/fake.py

[7]https://github.com/openstack/nova/blob/master/nova/virt/fake.py

[8]`https://wiki.openstack.org/wiki/Blueprints`

[9]A blueprint for service `nova-consolidator` is currently available at `https://blueprints.launchpad.net/nova/+spec/nova-consolidator`.

[10]http://docs.openstack.org/developer/nova/devref/filter_scheduler.html#weights

Bibliography

[1] E. Feller, C. Rohr, D. Margery, and C. Morin, "Energy Management in IaaS Clouds: A Holistic Approach.," *IEEE CLOUD*, pp. 204–212, 2012.

[2] A. Beloglazov and R. Buyya, "OpenStack Neat: a Framework for Dynamic and Energy-efficient Consolidation of Virtual Machines in OpenStack Clouds," *Concurrency and Computation: Practice and Experience*, 2014.

[3] J. G. Koomey, "Growth in Data Center Electricity Use 2005 to 2010," pp. 1–24, July 2011.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "A view of cloud computing.," *Commun. ACM ()*, vol. 53, no. 4, pp. 50–58, 2010.

[5] X. M. Zhang and N. Zhang, "An Open, Secure and Flexible Platform Based on Internet of Things and Cloud Computing for Ambient Aiding Living and Telemedicine," in *Computer and Management (CAMAN), 2011 International Conference on*, pp. 1–4, IEEE, 2011.

[6] N. Sultan, "Cloud computing for education: A new dawn?," *Int J. Information Management ()*, vol. 30, no. 2, pp. 109–116, 2010.

[7] K. Chard, S. Caton, O. Rana, and K. Bubendorfer, "Social Cloud: Cloud Computing in Social Networks.," *IEEE CLOUD*, pp. 99–106, 2010.

[8] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey.," *Future Generation Comp. Syst. ()*, vol. 29, no. 1, pp. 84–106, 2013.

[9] H. Goudarzi and M. Pedram, "Energy-Efficient Virtual Machine Replication and Placement in a Cloud Computing System.," *IEEE CLOUD*, pp. 750–757, 2012.

[10] X. Meng, V. Pappas, and L. Z. 0002, "Improving the Scalability of Data Center Networks with Traffic-aware Virtual Machine Placement.," *INFOCOM*, pp. 1154–1162, 2010.

[11] J. Xu and J. A. B. Fortes, "Multi-Objective Virtual Machine Placement in Virtualized Data Center Environments.," *GreenCom/CPSCom*, pp. 179–188, 2010.

[12] F. Wuhib, R. Stadler, and H. Lindgren, "Dynamic resource allocation with management objectives - Implementation for an OpenStack cloud.," *CNSM*, pp. 309–315, 2012.

[13] A. Corradi, M. Fanelli, and L. Foschini, "VM consolidation: A real case based on OpenStack Cloud.," *Future Generation Comp. Syst. ()*, vol. 32, pp. 118–127, 2014.

[14] S. He, L. Guo, M. Ghanem, and Y. Guo, "Improving Resource Utilisation in the Cloud Environment Using Multivariate Probabilistic Models.," *IEEE CLOUD*, pp. 574–581, 2012.

[15] H. Nakada, T. Hirofuchi, H. Ogawa, and S. Itoh, "Toward Virtual Machine Packing Optimization Based on Genetic Algorithm.," *IWANN*, vol. 5518, no. Chapter 96, pp. 651–654, 2009.

[16] X. Xu and H. Yu, "A Game Theory Approach to Fair and Efficient Resource Allocation in Cloud Computing," *Mathematical Problems in Engineering*, vol. 2014, pp. 1–14, 2014.

[17] P. Anderson, S. Bijani, and H. Herry, "Multi-agent Virtual Machine Management Using the Lightweight Coordination Calculus.," *T. Computational Collective Intelligence ()*, vol. 8240, no. Chapter 7, pp. 123–142, 2013.