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

Giacomo BRESCIANI

Matr. 804979

Lorenzo AFFETTI

Matr. 799284

RELATORE

Prof. Sam Jesus
Alejandro GUINEA
MONTALVO

Anno Accademico 2013/2014

dedica. . .

Ringraziamenti

Milano, 1 Aprile 2005

Affear

Table of Contents

List of Figures

List of Tables

List of Algorithms

Listings

Chapter 1

Introduction

Chapter 2

OpenStack and DevStack

In this chapter we are going to present OpenStack and DevStack giving a brief overview of them and focusing on their components and aspects that concern our thesis topic.

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 to control all of them trough a dashboard and enabling enterprises and service providers to offer on-demand computing resources.
One of its main strengths is the modularity that provides the necessary flexibility to design different configuration for a cloud environment; 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 number of Virtual Machines.

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

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

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

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

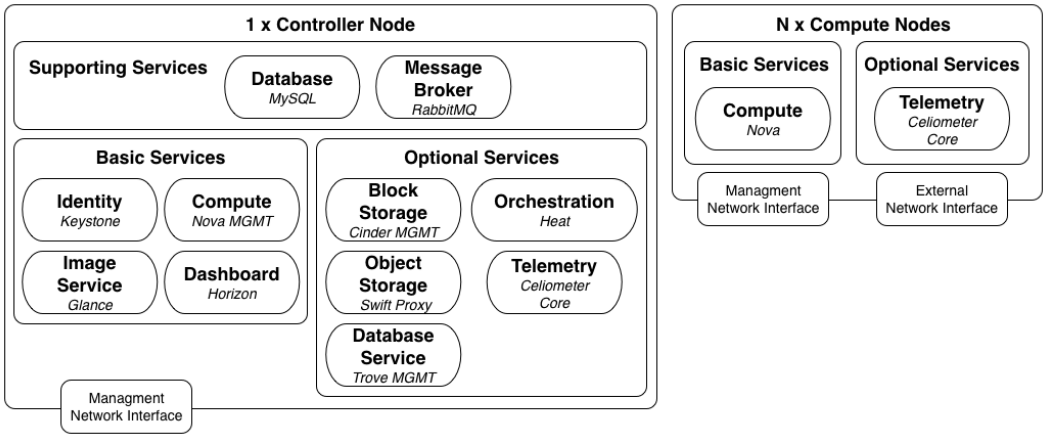

Figure 2.1: A "1 + N" OpenStack configuration

We have decided to focus on the creation of a "1 + N" installation of OpenStack that is composed by 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 6 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 to deploy and manage Virtual Machines, place them on physical machines, let them communicate, store their informations on an SQL database and offers both a set of APIs reachable through HTTP requests and a command-line client. In the next paragraphs we 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 to boot, resize, live-migrate and destroy Virtual Machines running on physical server and let 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 included directly in `Nova`. Unlike `Neutron`, that is able to virtualize and manage both layer 2 (logical) and layer 3 (network) of the OSI network, `nova-network` provides only simple layer 3 virtualization and has some limitations on the network topology.
However `nova-network` is still supported by OpenStack and can powerful enough to support a "1 + N" configuration needed for our purposes. 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 make the Virtual Machines placement decisions. The process of placing Virtual Machines on the more suitable host is divided in a *filtering* step where a list of candidate hosts is generated and a *weighing* step where the list is ordered according to the selected criteria and the best host is chosen.

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

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

(Nova), Network (Nova), Dashboard (Horizon) and Orchestration (Heat). The main script is `stack.sh`; all the configuration such as Git repositories to use, services to enable or OS images to use can be achieved overriding default environment variables (found in `stackrc`) by creating the file `local.conf` 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
# ...
```

The environment variable `ENABLED_SERVICES` is used to define the service to run: in listing 2.2 are shown the Nova services to install (`nova-compute`, `nova-api`, `nova-network`) in a simple compute node installation. 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

At the beginning of the development of the thesis we were mainly focused on the implementation of a module for OpenStack that would allow to implement different consolidation algorithms and test them to see their real impact on a cloud system in terms of resource allocation. During the first phases we faced with the problem of running, testing and benchmarking our code in an OpenStack environment: to deal with aspects like Scheduling, Virtual Machines Placement, and Server Consolidation we needed an highly configurable system that would allow us to run simulations and benchmark to evaluate the goodness of our solution.

A common barrier to experimenting with cloud infrastructure is in fact the lack of access to a fully functional cloud installation. 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. Moreover, setting up a testbed is necessary but not sufficient. One must also be able to create repeatable experiments that can be used to compare ones results to baseline or related approaches from the state of the art. So we designed and develop a system to address this problem: we wanted it to be fully customizable to match different requirements and let the user customize a lot of aspect

such as the structure of the environments, the number of Compute Nodes (the nodes that host Virtual Machines), their fake characteristics or the OpenStack services to run. Secondly we needed a way to automatically simulate, in a repeatable way, the workload generated from user applications that normally run on an OpenStack installation. At last we realized that it would be very useful to show the real time data of the simulations to analyze the behavior of the system in different configurations. Therefore we decided to develop aDock, a suite of tools for creating performance, sandboxed, and configurable cloud infrastructure experimentation environments that developer, sysadmins and researchers can exploit to access a fully functional cloud installation of OpenStack. For that reason this chapter is divided into two sections that present the state of the arts of Virtual Machines Consolidation and of Cloud test environments.

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 [**?**]. 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 [**?**], education [**?**], social networking [**?**], and mobile applications [**?**].

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 [**?**], [**?**]. Virtual Machine

(VM) Placement [?], [?] defines how a cloud installation decides on which physical server to create a new virtual machine, when one is requested. Server Consolidation techniques [?], [?], 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 clouds customers.
Dynamic consolidation of Virtual machines is enabled by *live migration*that is the capability of moving a running Virtual Machine between two physical hosts with no downtime and no disruptions for the user. Thanks to dynamic Virtual Machines consolidation is therefore possible to live migrate Virtual Machines from underutilized hosts to minimize the number of active hosts and remove Virtual Machines from hosts when those become overloaded to avoid performance degradation.

With regard to Virtual Machine Consolidation a lot of solutions, algorithms and techniques were proposed in literature [?], [?], [?], all with different approaches to the problem; 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 attempted at 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* [?] 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 an heuristic algorithm a type of techniques that are often used to address NP-hard problems as the bin packing problem. A GA is a model of machine learning that takes inspiration from the concept of evolution observed in biological environment from which it borrows a lot of terms such as Chromosome, Mutation or Pop-

ulation.

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 used a One-Point Crossover that randomly cut two chromosomes and mix them. They implemented a repair function to fix the inconsistent children thus obtained.

Mutation They randomly exchange two position between them.

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

Objecive Function The unspecified objective function is said to be designed with some 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 simulations tests are not described in a detailed way and there are no data results to prove the goodness 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 this 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 **#TODO** aggiungere riferimento.

3.2.2 Holistic Approach

The paper *Energy Management in IaaS Clouds: A Holistic Approach* published during the Fifth International Conference on Cloud Computing of IEEE in 2012 presents the energy management algorithms and mechanisms of holistic energy-aware Virtual Machines management framework for private clouds

called Snooze.
The system architecture described by the authors (see figure 3.2.2 on page 16) is divided in three layers:

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

- *Local Controller* - They enforce Virtual Machines and host management commands coming from the GM. Moreover, they monitor VMs, detect overload/underload anomaly 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 retrieving their resource information and sending 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 GL level, where VM to GM dispatching is done based on the GM resource summary information in a round-robin way, and GM level where the real scheduling decisions are made. In addition to the *placement policies*, applied when a new VM is requested, are present *relocation policies*, called when overload or underload events arrive from LCs, and *consolidation policies*, called periodically according to the system administrator specified interval.
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[**?**, p. 208].

In section **#TODO** sezione.. we describe how we implemented this algorithm in our system and the result obtained with our configuration.

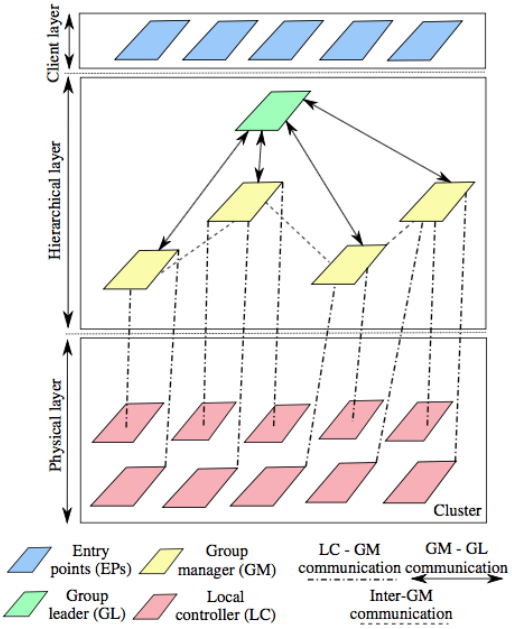

Figure 3.1: The Snooze architecture [**?**]

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 [**?**].

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

CEM - Cloud Environments Monitor This component retrieves information like host names, IP addresses about physical servers, 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 computing center to provide the most appropriate decision about resource allocating.

CEM is monitoring the statuses and resource consumptions for physical servers registered in RC. Once a new physical server started to join the cloud, the information like MAC address, IP address will be registered to RC. When a user sends a service request to cloud, the requirements of resources in 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.[**?**, p. 3]

They experiment a FUGA (Fairness-Utilization tradeoff Game Algorithm) on a server cluster composed by 8 nodes and show that it make it possible to

achieve an optimal tradeoff between fairness and efficiency compared with the evaluation of the Hadoop[1] 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* is based on an LCC system that specify the migration behaviour of Virtual Machines within, and between cloud environments. LCC is a Lightweight Coordination Calculus which can be used to provide an executable, declarative specification of an agent interaction model[**?**].

The proposed system is a distributed between nodes and so doesn't have a central controller that could represent a single point of failure or a bottleneck: agents located on the physical machines negotiate to transfer VMs between themselves, without reference to any centralized authority[**?**, 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 several different situations.

3.2.5 Neat

OpenStack, at the state of the art, provides a comprehensive and efficient Virtual Machines Placement system found, as described in chapter 2 **#TODO** inserire la sezione giusta, in the `nova-scheduler` module; however, with regard to Virtual Machines Consolidation, OpenStack doesn't include any official solution or plans to include it.
The only project that tried to bring the Virtual Machine consolidation concepts to OpenStack is Neat[2], it is defined as a framework for dynamic and energy-efficient consolidation of virtual machines in OpenStack clouds [**?**].

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

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

- Deciding if a host is considered to be *underloaded*, so that all Virtual Machines should be migrated from it, and the host should be switched to a low-power mode.
- Deciding if a host is considered to be *overloaded*, so that some Virtual Machines should be migrated from it to other active or reactivated hosts to avoid violating the QoS requirements.
- Selecting Virtual Machines to migrate from an overloaded host.
- Placing Virtual Machines selected for migration on other active or reactivated hosts.

In figure 3.2.5 [**?**, p. 7] it is represented the architecture of OpenStack Neat: it is mainly composed by a *Global Manager* installed on the Controller node, a *Local Manager* and a *Data Collector* both installed on every Compute node. The *Global Manager* is responsible for making global management decisions such as mapping Virtual Machines instances to hosts, and initiating Virtual Machines live migrations; the *Local Manager* makes instead 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 data on the resource usage by Virtual Machines instances and hypervisors and 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 processes by invoking public APIs of OpenStack. For that reason it has to be installed separately from OpenStack, following the limited instructions present on the GitHub page of the project[3] and can not take advantage of tools like

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

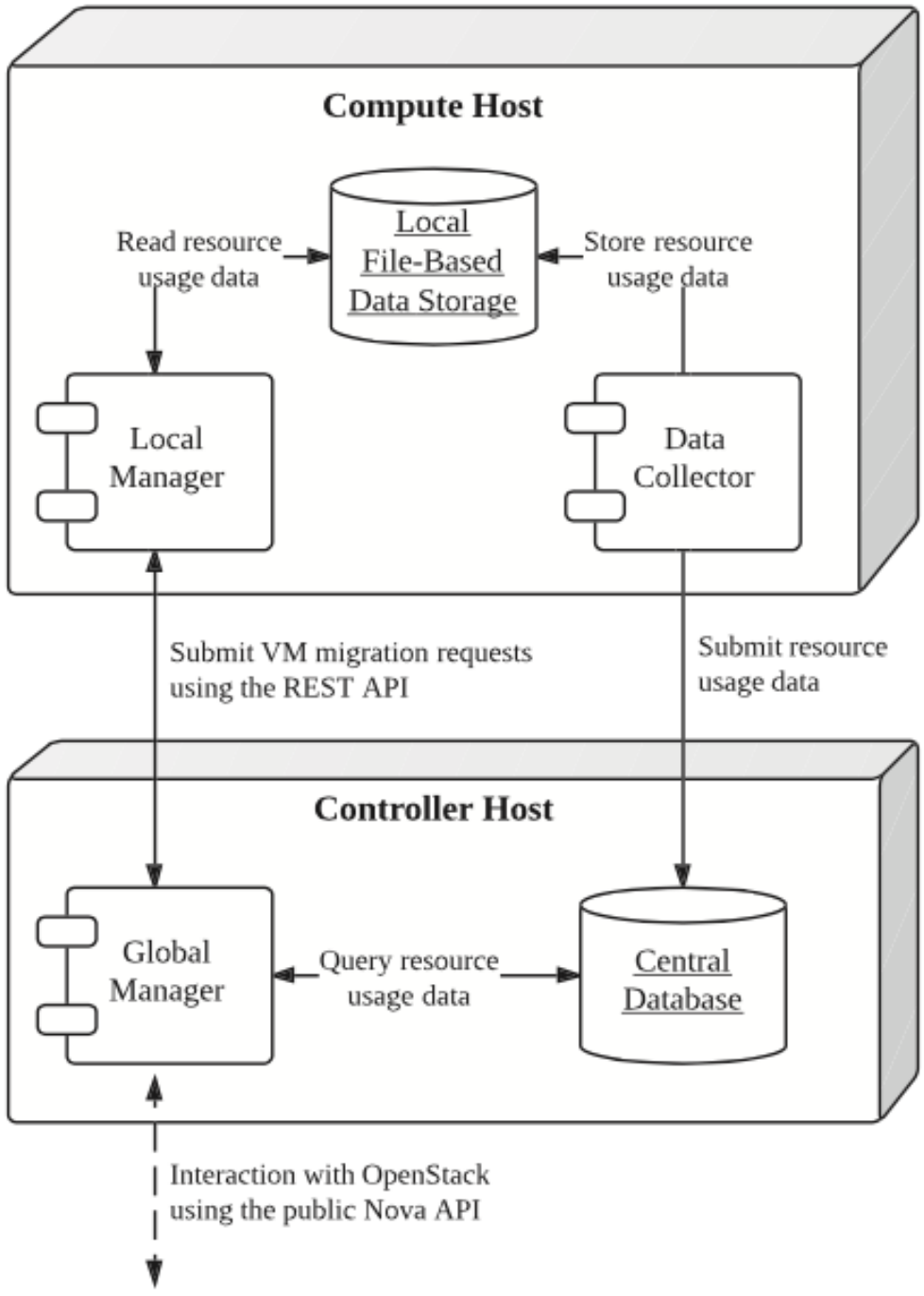

Figure 3.2: The OpenStack Neat architecture [**?**]

DevStack (see 2 on page 5) that automates the deploy and configuration of an OpenStack installation.

3.3 Cloud test environments

In the cloud world is fundamental, as in any engineering field, to be able to test environment configurations and algorithms, both to analyze the behavior of tested code that integrates with a real environment and to benchmark and collect data for researches 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 understand and handle an OpenStack installation is not easy, especially for non sysadmins **#TODO** (Find a better way to say it) like developers or researchers, it has indeed an high learning curve and often it is necessary a lot of time to achieve a good and desired result. To reduce the impact of these complications are available some tools that make the process of setting up a cloud infrastructure experimentation environment more easy and manageable. For these reasons and for the growing need of advanced system management 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 are presenting them highlighting their main features and their

strengths and weaknesses with respect to our thesis topic.

3.3.1 Vagrant

Vagrant[4] is a virtualization framework for creating, configuring and managing development environments, written in Ruby, that allows to virtual development environments. 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 an online repository [5] 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:

```
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
```

[4] www.vagrantup.com

[5] www.vagrantcloud.com

Listing 3.1: `Vagrantfile` example

Provisioners in Vagrant allows to automatically install and configure software in a Vagrant Box as part of the "vagrant up" process, therefore you can 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 in combination with configuration management software such as Chef and Puppet is used to create repeatable and easy to setup development and test environments that rely on Virtual Machines.

3.3.2 Chef

Description Chef[6] is a configuration management tool used to streamline the task of configuring and maintaining servers in a cloud environment and 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)[7] for writing configuration files called "recipes". "Recipes" are used to define in a declarative way the state of certain resources and define everything that is required to configure different parts of the system: they can contain the definition of software that should be installed and all the required dependencies, services that should be running 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[8], that collects a large number of "cookbooks" from the community freely downloadable.
A base installation of Chef is composed by three main components, a

[6] `www.chef.io`

[7] A programming language specialized to a particular application domain.

[8] `supermarket.chef.io`

`chef-server` that orchestrates all the Chef processes, multiple `chef-client` found on all the servers, and the user workstation that communicates with the Chef Server to launch commands.
To simplify the communication with the `chef-server` Chef provides a command-line tool called Knife that helps users to manage nodes, "cookbook" and "recipes", and the majority of possible operations.

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. Is it 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 thus instantiate new physical servers with a chef-client installed or turn them off (`knife openstack server create / delete`). In this situation you can achieve a "1 + N" configuration that is one OpenStack Controller and N OpenStack Nodes, and the OpenStack services are predefined and you cannot configure an ad hoc configuration. Therefore an "All-in-One" configuration can be chosen 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[9] there is a "recipe" that 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 Vagrantfile to create and configure one VM to host the Controller and N VMs to host the Compute nodes. However it is unlikely to succeed in running a lot of VMs on the same host especially if they will contain a fully functional

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

```
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`)

OpenStack installation as a Virtual Machine typically require a significant amount of resources to operate.

#TODO ... Insert timing to install OpenStack with Chef...

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

- *Heaviness*: due the greed of resources of a Virtual Machine is very difficult to achieve a "1 + N" configuration for development or research purpose on 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 as it runs all the OpenStack services on a single node.

- *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 service (in particular are installed all the OpenStack service excluded Object Storage, Metering, and Orchestration) so it is

not possible to setup the environment with more or less services or new ones (as in our case).

- #**TODO** think others...

3.3.3 Puppet

Description Similarly to Chef (described in section 3.3.2) Puppet[10] is a configuration management system that allows you to define the state of a cloud infrastructure and then it automatically enforces the correct state.
Puppet uses a declarative model where are defined the resource states and (likewise Chef) it's manifest files are written in a Ruby-like DSL. In these manifest files are defined the configurations, the nodes and how the configurations apply to nodes. Again Puppet will take care of ensuring that the system reaches the expected state. All these files are enclosed in "modules", a self-contained bundles of code and data easy to share and reuse. There are a large amount of them on Puppet Forge[11] repository.
Puppet is structured in master-slave architecture: the master (that can be one or machines) 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. This structure uses the "Puppet master" and "Puppet agent" applications.

Puppet and OpenStack As seen for Chef, Puppet can be very useful when dealing with OpenStack installation and maintenance. To configure and deploy an OpenStack infrastructure with the help of Puppet exists a set of "modules" freely downloadable from Puppet Forge that simplifies most of the operations such as OpenStack instances provisioning, configuration management and others. The module is `puppetlabs-openstack`[12] and allows to

[10] `www.puppetlabs.com`
[11] `forge.puppetlabs.com`
[12] `github.com/puppetlabs/puppetlabs-openstack`

deploy both a multi-node and an all-in-one installation. Compared to Chef, with regard to OpenStack, Puppet is a bit more flexible because it allows you to control in more details the OpenStack services installed on every node; for example you can, combining the following instructions in the Puppet's manifest file of a node, achieve different results:

Controller node:

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

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

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

Obviously, in the same way for Chef, it is possible to configure multiple nodes to run in multiple Virtual Machines configured and launched with Vagrant and deploy the various OpenStack components with `puppetlabs-openstack`. This solution, as said earlier, is difficult to achieve on a machine with a limited amount of resources, and also on more powerful server is however slightly extensible.

Pro and Cons Puppet is an extremely powerful and mature tools for automated cloud infrastructure deploying: it streamlines the entire process automating every step of the software delivery process.
However from our point of view it is more relevant to how it behave when a single developer or a researcher needs to deploy a cloud infrastructure within a single machine with limited amount of resources (a development workstation for example) or he/she has a little sysadmin skills (**#TODO** find better way) and want to run test for developing or researches purposes. With regard to this aspect Puppet used with Vagrant has some key limitations:

- *Heaviness*: A single Virtual Machine need generally a remarkable amount of resource, especially to host an OpenStack installation; for this reason it is very unlikely to be able to run the number of Virtual Machines needed to deploy a realistic multi-node installation of OpenStack on a single machine without compromising its usage. The all-in-one instead is frequently not sufficiently realistic, especially when testing algorithms or portion of code that involves multiple nodes.
- **#TODO** think others...

3.3.4 Docker

Docker[13] 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 an host one avoiding the considerable amount of resources necessary to run Virtual Machines.
The main difference between the Docker solution and the Virtual Machine one lies 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 29, hosts a complete Operating System including application, dependency libraries, and, more important, the kernel; otherwise the Docker Engine runs

[13] `www.docker.com`

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 portable and efficient; for our goals this aspect represents the possibility 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 **#TODO** Better way... .

To configure and then 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 that will be almost instantly running.

Moreover Docker offers an online platform called Docker Hub[14] where you can upload both Dockerfile and pre-built container images to streamline the sharing process.

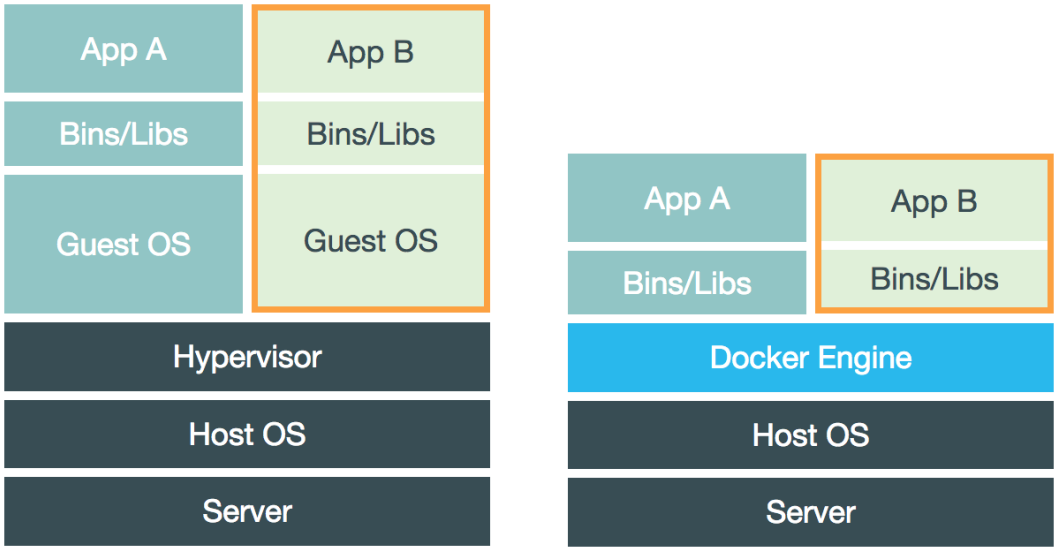

Figure 3.3: Hypervisor and Docker Engine

[14] `hub.docker.com`

3.3.5 Dockenstack

One of the first an attempts to create a cloud test environments based on OpenStack and Docker is a Dockenstack[15]. It is an independent and not actively supported project, but is a good starting point to show the potential derived from using Docker.

The project is basically composed by a Dockerfile and a bunch of scripts that will setup and configure an OpenStack installation using DevStack (**#TODO** shall we talk about it in the intro?) in a Docker container. **#TODO** cmd spiegare

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 Machines placement and server consolidation algorithms.

[15] `github.com/ewindisch/dockenstack`

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 cloud computing software platform, because of its open-source nature and because of its continuous evolution with the aim of keeping up with the last 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 algorithm on a complete cloud system to test out its behavior.

4.2 Requirements

In this section, we will identify both functional and non-functional requirements for aDock.

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. aDock should provide the user with an abstraction of a server called "node". The "node", in its depth, is an Ubuntu based Docker container, shipped with OpenStack services dependencies. A user should be able to decide which services will be installed and started on each node and their internal configuration using a configuration file.

Solution FakeStack (see section 4.3) is the aDock module which provides the user with the specified tools. 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.

FR2 *aDock should provide a tool to run simulations.*
If the user puts his/her code into OpenStack he/she probably aims at running simulations and examine the new piece of code behavior in interacting with the entire system. Simulations should be configurable according to the user needs and repeatable.

Solution Oscard (see section 4.4) is the aDock module which takes care of running repeatable and configurable simulations against an OpenStack system.

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

Solution Oscard, 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).

FR4 *aDock should provide a user interface*
Although Firebase provides an interactive user interface, data is displayed in a *JSON* fashion and it is, therefore, not easily understandable and browseable. Simulations results should be displayed to user in a friendly manner, using charts to give the user a glimpse of the current situation. Data representation should be given in real-time.

Solution Polyphemus (see section 4.5) is the aDock module which takes care of displaying to the user real-time simulation results in a friendly manner.

aDock turns out to be 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 results obtained. In figure 4.2.1 we highlight the general architecture of aDock.

4.2.2 Non-Functional Requirements

As opposed to functional requirements, aDock has very strong non-functional requirements in order 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 high speed in running containers and sandboxing by construction and makes aDock cross-platform; while DevStack gives us great flexibility and configurability for what concerns OpenStack services.

NFR1 *Users should be able to choose which code is running in OpenStack.*
Before booting the entire system the user should be able to choose if he/she

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

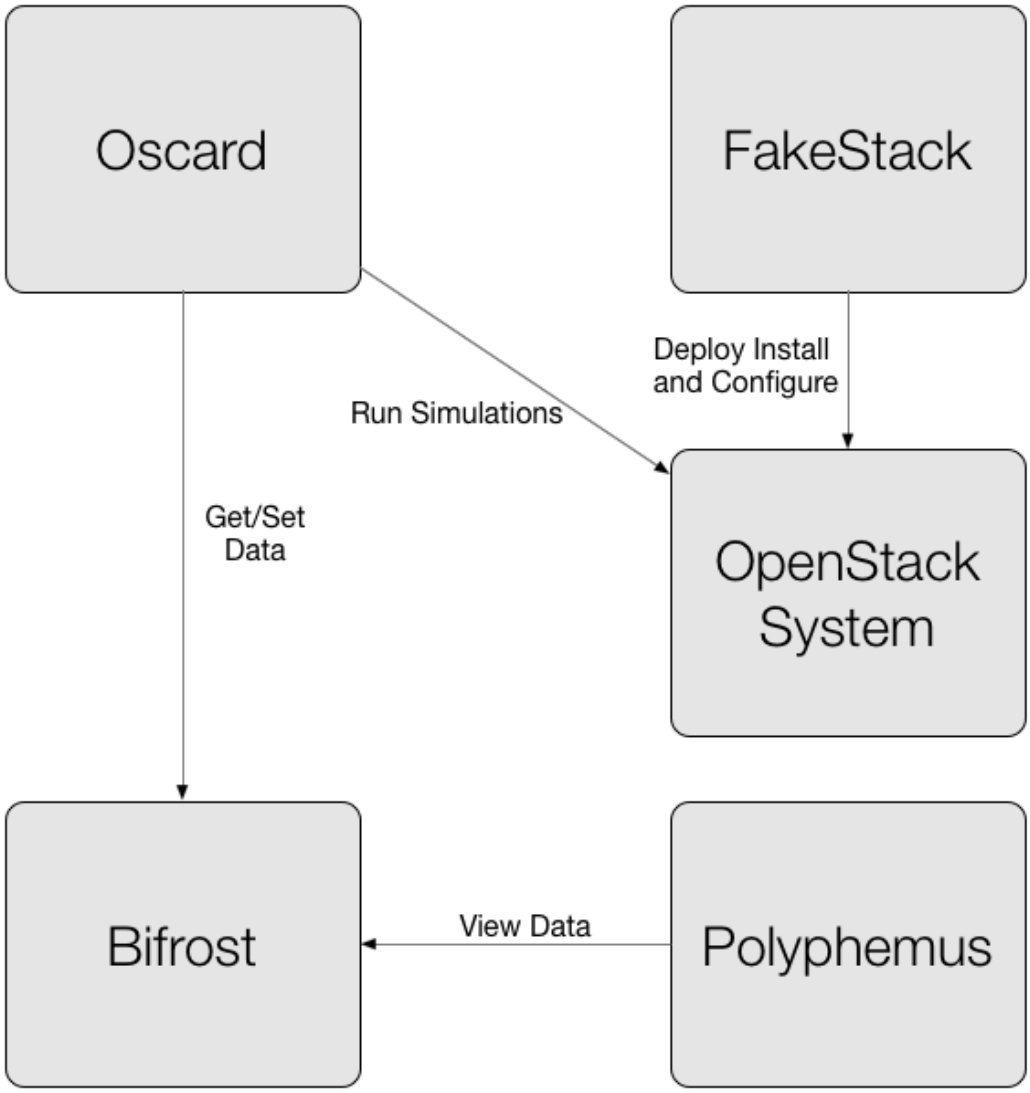

Figure 4.1: aDock high level architecture

wants to run OpenStack code from a precise code repository which is, in general, the better and more supported way to version and share code among developers and even physical machines. 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

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

(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 onto nodes.

Solution All of this is achievable thanks to DevStack, which installs OpenStack services 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.

Solution Docker is a virtualization system which relies on Docker containers which are much more lightweight than virtual machines[3][4] **#TODO** are those refs authoritative? . 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

[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/`

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 OpenStack in all its aspects through `local.conf` file. Each service can be configured in each of DevStack 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:

```
1  ... # DevStack configurations
2
3  [[post-config|\$NOVA_CONF]]
4  [DEFAULT]
5  verbose=True
6  logdir=/var/log/my-nova-logdir
7
8  # SCHEDULER
9  compute_scheduler_driver=nova.scheduler.MyMagicScheduler
10 # VIRT DRIVER
11 compute_driver=nova.virt.fake.MyAmazingFakeDriver
```

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

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.

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

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 which allows the user to manage *nodes*, which are 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.

4.3.1 Nodes

As already anticipated in requirement 4.2.2, 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 the presence, in the first one, of database (in our case, *MySQL*) and message broker (in our case, *RabbitMQ*) services, compulsory for OpenStack's controller nodes.

[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.

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

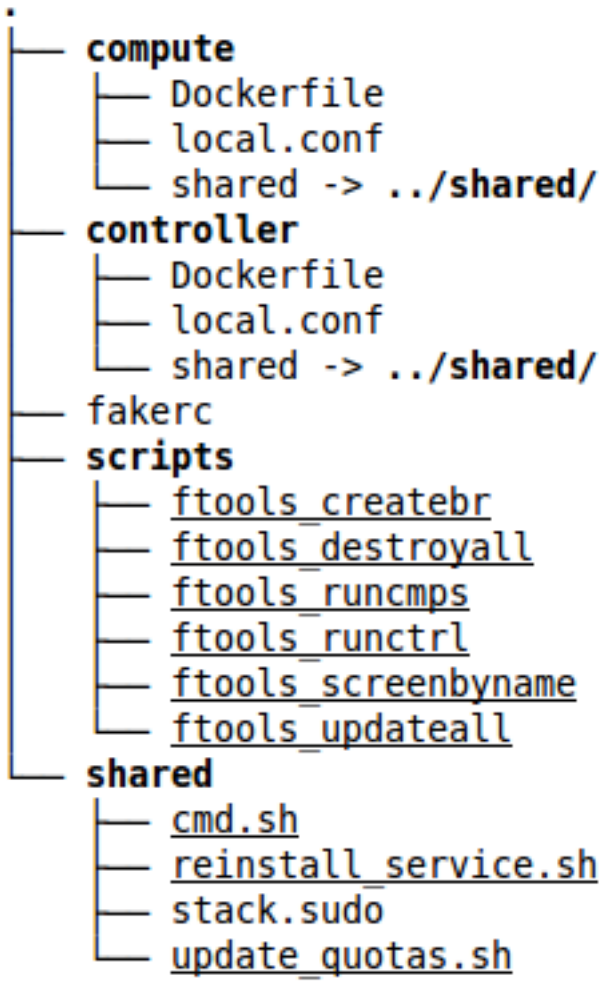

Figure 4.2: FakeStack's file structure

As we can see in figure 4.3.1, nodes have two separated `Dockerfile`s (which makes them two different Docker containers), but they share a set of scripts contained in `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 probably the user will spawn hundreds or thousands of virtual machines on its OpenStack nodes and, normally, standard quotas will prevent him from doing so. Enlarging quotas is an easy and fast way to allow the user not to worry about how many virtual machines he owns.

Once a node has been built, all of this shared scripts are copied to the file-system of the node.
`local.conf` file, instead, is bound[10] to node's file-system avoiding 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] `local.conf` is a *Data Volume*. Basically it is a file whose modifications are visible at each container's run.

When a node is started, `cmd.sh` will run, resulting in running DevStack's `stack.sh`. At this very moment OpenStack's installation starts **??** #**TODO** reference to DevStack's detailed description.

4.3.2 Scripts

Once a user enters FakeStack root directory he/she has to `source fakerc`. Executing this command, all of the scripts contained in `scripts` directory become available in `PATH`. The prefix `ftools_` is added to avoid conflicts in names.

Scripts for running and updating nodes takes leverage of Linux screens[11]. Screens are powerful tools 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 be obliged to open lots of shells, but he could use only one; keep it clean from computation and reattach to aDock screens once 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 opened and wait for simulations to end; the screen session, in fact, will stay open (and so the processes within it, running) indipendently from SSH connection.

We list here and describe scripts contained into `scripts` directory.

createbr Input: bridge name. Creates a bridge with CIDR 42.42.0.0/24 named as the bridge 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 edit IP configuration in `cmd.sh`. In fact, `cmd.sh` will set controller's IP to the last available into that precise net-

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

work (42.42.255.254). After running this script, Docker service has to be restarted.

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 list here and describe scripts "sourced" from `fakerc` (`ftools_` convention is always maintained).

runcmp Alias for `ftools_runcmps 1`.

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

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

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

4.3.3 Configuration

#TODO repair listings placement!

Fakestack takes leverage of the powerful configurable options of DevStack. Modifying `local.conf` files before starting a node, it is possible to change enabled services (see listing 4.1) and their internal configuration (see listing 4.2). Every service is configurable in each of its installation *phases*. 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. DevStack's service installation phases are **local**, **pre-install**, **install**, **post-config**, **extra**[13].

Most important, it is possible to choose a different Git repository and Git branch for each of OpenStack's services enabled (see listing 4.3). 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 4.4 we show a possible complete example of `local.conf` file for a compute node.

```
... # Other configuration options

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

[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
```

Listing 4.1: Choose OpenStack's enabled services

```
... # Other configuration options

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

... # Other configuration options
```

Listing 4.2: Internal configuration of Nova

```
... # Other configuration options

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

... # Other configuration options
```

Listing 4.3: Change repository URL

```
1  [[local|localrc]]
2  FLAT_INTERFACE=eth0
3  MULTI_HOST=1
4  LOGFILE=/opt/stack/logs/stack.sh.log
5  SCREEN_LOGDIR=$DEST/logs/screen
6
7  NOVA_REPO=https://github.com/me/nova.git
8  NOVA_BRANCH=my-branch
9
10 DATABASE_TYPE=mysql
11
12 ADMIN_PASSWORD=pwstack
13 MYSQL_PASSWORD=pwstack
14 RABBIT_PASSWORD=pwstack
```

```
15 SERVICE_PASSWORD=pwstack
16 SERVICE_TOKEN=tokenstack
17
18 SERVICE_HOST=controller
19 MYSQL_HOST=controller
20 RABBIT_HOST=controller
21
22 NOVA_VNC_ENABLED=False
23 VIRT_DRIVER=fake
24
25 ENABLED_SERVICES=n-cpu,n-api,n-net
26
27 [[post-config|$NOVA_CONF]]
28 [DEFAULT]
29 compute_driver=nova.virt.fake.MyFakeDriver
```

Listing 4.4: 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 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 which takes care of running simulations against one or more OpenStack systems and collect data output from them. Oscard has two principal components, a server and a client one (see 4.4.2). The two components don't need to be used on the same machine. It is for these reason that Oscard's dockerized version runs only the server part and waits for requests from the client.

The client part is the one that actually runs simulations. A user can configure a simulation in terms of number of steps, command weights and more and run it using the script provided.

The server part is the *Proxy*, it litterally waits for client requests and forwards them to OpenStack's controller node.

Oscard is completely configurable from `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.

oscard.sim.api This module contains the APIs to interact with OpenStack's Nova. It mainly contains two classes, `NovaAPI` and `FakeAPI`. `NovaAPI` provides 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 into `oscard.conf`, the new seed will be reloaded as well.
- `architecture`: Returns system's architecture in terms of compute nodes and their resources (vCPUs, memory and disk).

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

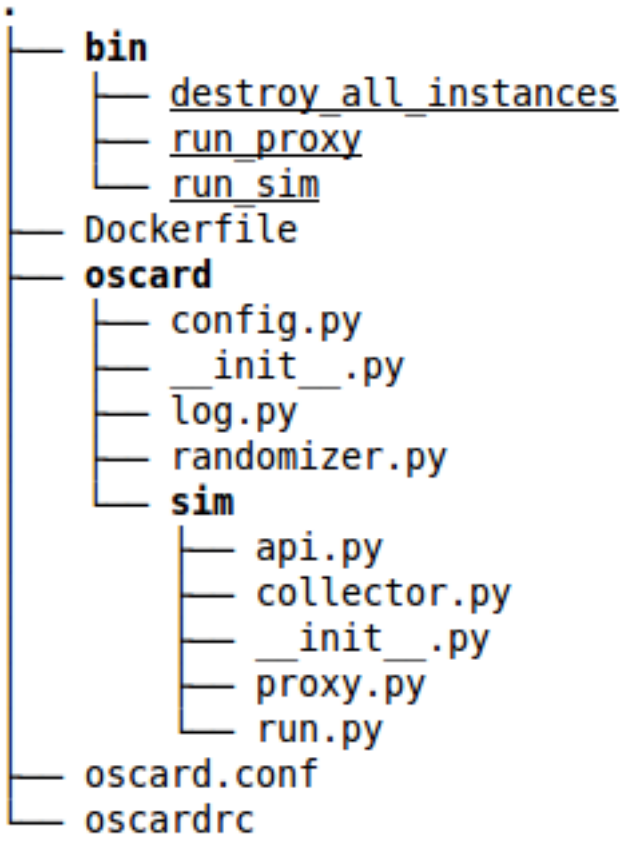

Figure 4.3: Oscard's file structure

- `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 hosting 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 `BifrostAPI` class, which gives the APIs to interact with the Firebase backend for data storage. An instance of it is obtained in `oscard.sim.run` module and it 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 every method contained in `oscard.sim.api.NovaAPI`. A `NovaAPI` object is obtained at module startup and each of the calls to proxy's methods is *delegated* to it. 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 in detail Oscard internal working and its possible configuration options (specified in footnotes, when necessary). We will also clarify some of the concepts (such as pseudo-randomness) about simulations.

Each random-taken 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 by configuration file[17] or, in case it 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

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

[17] In `oscard.conf`: `random_seed`

reason that every random-taken decision can be repeatable simply setting the seed from 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. Available commands (up to now) are *create*, *resize*, *destroy* and *NOP*. First three commands are clear in their intent, the last one, *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 (with success or not), thus, the state of the machine interested in the operation, can be one of `ACTIVE` or `ERROR`[21] and not an intermediate one. 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*. At least, for what concerns Oscard itself: the system, in fact, is built to run repeatable simulations, but we cannot guarantee that OpenStack system will always take the same decisions. Thus, two simulation outputs could be different even if simulations are, for Oscard, the same one.

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[22]. Oscard, as FakeStack, has a source

[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`

[22] Dockerized version is recommended, because it allows the user not to install all Oscard's

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`). So, it is not needed to configure each Oscard option before starting it. Some options are relevant only for server, others for client and some for both of them. We list their meaning and split them between the two Oscard components to better understand their working.

The server part can be configured in terms of:

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

dependencies before running

- `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).

As an example of Oscard functioning, we show in figure 4.4.2, by means of a sequence diagram, the workflow of a `create` operation.

#TODO sequence diagram of create op

Figure 4.4: Workflow for a `create` operation

4.5 Other Components

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

4.5.1 Bifrost

Bifrost[23] is the name for the Firebase application delegated 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

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

our case and because of its cloud nature. It is important that Firebase, by design, doesn't offer a great support for concurrent calls[24], so, our simulations *cannot* run concurrently[25].

Firebase offers a good dashboard (see 4.5.1) that updates in real-time (if the amount of data is not too big) and allows the user to perform CRUD[26] operations on data.

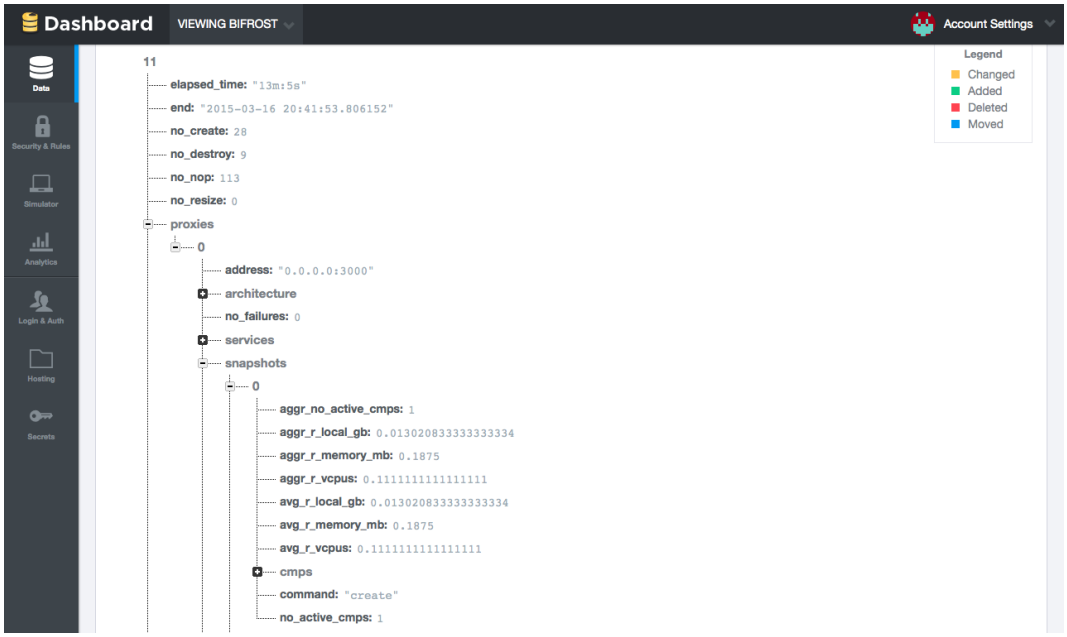

Figure 4.5: Firebase Dashboard

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

[25] The same simulation can be run concurrently on more hosts, but two simulations cannot be run concurrently on different hosts.

[26] Create Read Update Delete

4.5.2 Polyphemus

Polyphems[27] is the Polymer[28]-powered view of aDock. It can be run in a Docker container or not[29]. Its aim is to show data to the user in a friendly way. Polyphemus shows each simulation snapshot in terms of overall average of resources usage[30] through line charts and percentage of resources usage[31] for each compute node through bar charts. A tool-bar representing data aggregates[32] 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 make comparisons intuitively. Moreover it includes more information such 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.

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

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

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

[30] 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).

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

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

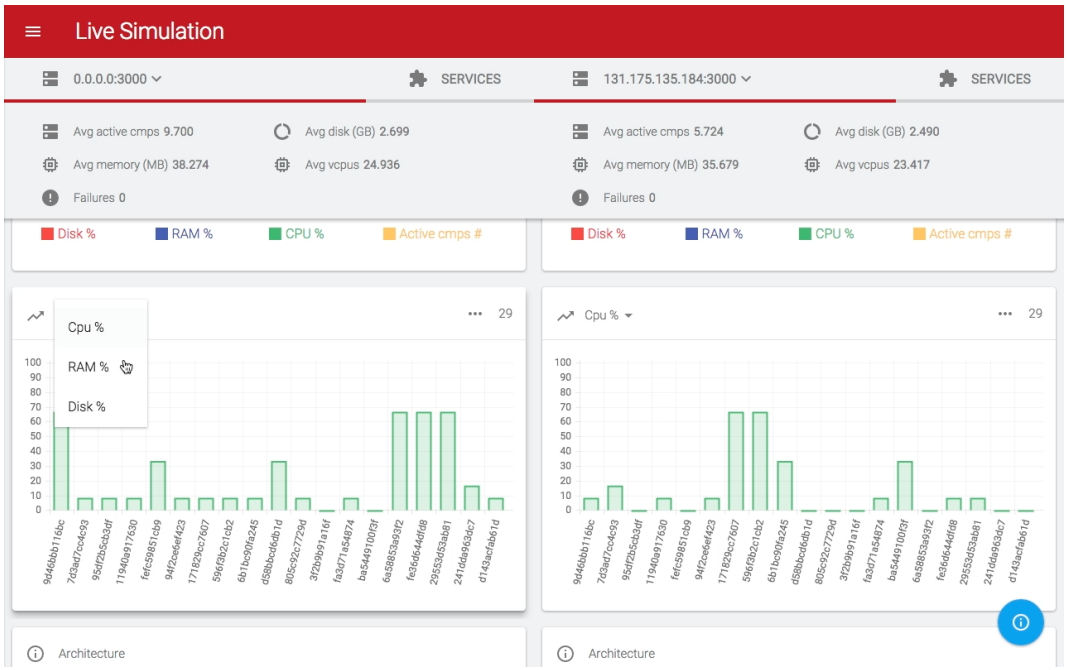

Figure 4.6: Screenshot of Polyphemus

4.6 aDock's Architecture

aDock is a modular system, where each component is run in its dockerized version[33]. In figure 4.6, a sample aDock architecture is shown.

The simulation is started from a normal laptop using Oscard. Oscard client contacts Oscard proxies on each of the hosts (specified in `proxy_hosts`) using the endpoints exposed, 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`) which, in turn, will handle it. For each host and at each step, the proxy collects data from the controller using `NovaAPI` methods and stores it into Bifrost using `BifrostAPI`. Data is available and can be consulted connecting to Polyphemus[34] using a web browser on user's laptop.

[33] Apart from Bifrost.

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

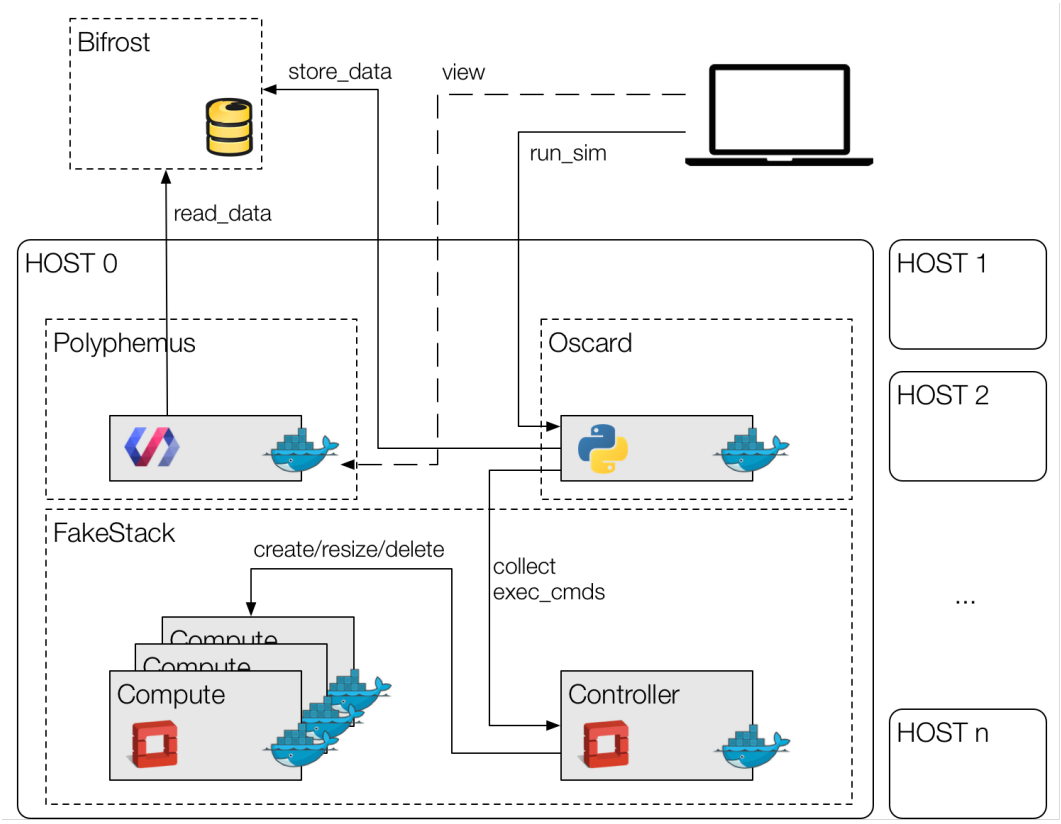

Figure 4.7: Screenshot of Polyphemus

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 *doesn't* 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 live (or cold) migrated to other hosts if some policy determines that its place is the wrong in that precise moment. The policy adopted is determined by the *consolidation algorithm* used.

To add virtual machine consolidation feature to OpenStack we added a service to `Nova` called `nova-consolidator`. The new ser-

[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.

vice is implemented in module `nova.consolidator` which provides a `nova.consolidator.base.BaseConsolidator` class which can be extended to write custom consolidators (see section 5.1) and some consolidation algorithms, both custom and taken from the state of the art (see section 5.2).

5.1 Consolidator Base

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

The command basically instantiates a `nova.service.Service` object with name "`nova-consolidator`". The service, in turn, instantiates a `nova.consolidator.manager.ConsolidatorManager` object; starts its RPC server and its *periodic tasks*. As we can see in listing 5.1[9], `ConsolidatorManager` exposes only one periodic task which is `consolidate` method. Its period is defined in `/etc/nova/nova.conf` file (which can be edited using DevStack. See 4.3.3) in option `consolidation_interval` as well as the consolidator class used by the manager. When the manager is created a consolidator object is obtained using the `consolidator_class` provided. Then, periodically, `consolidate` task is invoked. `consolidate` function *delegates* consolidation to the consolidator object obtaining the necessary

[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.

migrations to be performed. Once 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 defining a base class to be extended with real consolidation algorithms. Its `get_migrations` method, in fact, returns an empty list of migrations. The most important method in `BaseConsolidator` is `consolidate`, which is the method the manager delegates consolidation to.

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

5.1.1 Objects

We thought that it wouldn't have been fair to leave to the user the duty to learn and understand OpenStack's complex database APIs. Due to this fact, we developed `nova.consolidator.objects`, which is a module that defines an abstraction of system snapshot to be used in method `get_migrations` by developers. The module provides the class `nova.consolidator.objects.Snapshot`. A `Snapshot` object offers attributes to access all information about the system, such as current active nodes and instances both for each node and, generally, in the system itself. The `Snapshot` is thought to be renewed at each consolidation cycle. So, any attribute is lazily obtained on its first call: subsequent invocations of that attribute won't refresh snapshot's state. The Snapshot is, thus, entirely cached[11].

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

[11] Once an instance or a compute node is obtained it will not be queried again on OpenStack's database. Its status is *frozen* at the moment the first query has been performed. To refresh a `Snapshot` it is necessary to create a new `Snapshot` object.

```
1  class ConsolidatorManager(manager.Manager):
2
3    def __init__(self, *args, **kwargs):
4      self.compute_api = compute_api.API()
5      self.consolidator = importutils.\
6        import_class(CONF.consolidator_class)()
7      # lines skipped
8
9    @periodic_task.\
10     periodic_task(spacing=CONF.consolidation_interval)
11   def consolidate(self, ctxt):
12     migrations = self.consolidator.consolidate(ctxt)
13     for m in migrations:
14       self._do_live_migrate(ctxt, m)
15
16   def _do_live_migrate(self, ctxt, migration):
17     instance = migration.instance
18     host_name = migration.host.host
19     # exception catching skipped
20     self.compute_api.live_migrate(
21       ctxt, instance,
22       False, False, host_name
23     )
```

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

```
1  class BaseConsolidator(object):
2
3    class Migration(object):
4      def __init__(self, instance, host):
5        super(BaseConsolidator.Migration, self).__init__()
6        self.instance = instance
7        self.host = host
8
9    # _transitive_closure method
10   # implementation skipped
11
12   def consolidate(self, ctxt):
13     snapshot = Snapshot(ctxt)
14     migs = self.get_migrations(snapshot)
15     return self._transitive_closure(migs)
16
17   def get_migrations(self, snapshot):
18     return []
```

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

In detail, a `Snapshot` object offers active nodes (`nodes` attribute) and all, running, migratable[12] and not instances. Instances are `nova.objects.instance.Instance`[13] objects; nodes are wrappers for `nova.objects.compute_node.ComputeNode`[14] objects, which add the possibility to get all, running, migratable and not instances per compute node.

In any case, the developer is not thought to instantiate `Snapshot` objects, because this is up to `consolidate` method, which already instantiates and passes the current system snapshot to method `get_migrations`, which is the *only* method which 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 in detail consolidation algorithms that we implemented in our `nova-consolidator`. Each of the algorithms proposed is run inside a consolidator class that inherits from `nova.consolidator.base.BaseConsolidator`, inside `get_migrations` method.

5.2.1 Random Algorithm

The first algorithm we implemented is the random one[15]. This algorithm was implemented for testing purpose and to see if even randomization could bring

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

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

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

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

```
1  from nova import config, objects, context
2  from nova.consolidator.objects import Snapshot
3
4  # Init operations
5  config.parse_args('')
6  objects.register_all()
7  ctxt = context.get_admin_context()
8
9  # Using the Snapshot
10 s = Snapshot(ctxt)
11 nodes = snapshot.nodes # all compute nodes
12 node = nodes[0] # the first node
13 instances = node.instances # all instances on that node (list)
14 print node.vcpu
15 print node.id
16 print instances[0].flavor
17 # `node` has all attributes as
18 # nova.objects.compute_node.ComputeNode has,
19 # as well as `instances[0]` has all attributes as
20 # nova.objects.instance.Instance has.
21
22 nodes_new = snapshot.nodes
23 # nodes are not refreshed because they are cached!
24 assert nodes == nodes_new # evaluates to True
```

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

improvement in resource optimization, given that virtual machines are never moved in OpenStack[16].

The algorithm takes, by configuration, a percentage of migratable instances to be migrated to other compute nodes. Instances are randomly chosen from hosts and their destination is randomly chosen among remaining hosts. Choices are not taken taking into account host suitability. The algorithm by itself 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.

Random algorithm is highlighted in in listing 5.4[9].

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 heavily revisited from us[17].

Our genetic algorithm uses a list as chromosome structure. Each element of the list (a gene) is considered as a migratable instance an 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. After some simulation we realized that 100% of the children generated were unhealthy[18]. 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. While in crossover we chose[19] two chromosomes,

[16]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.

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

[18]A child is considered unhealthy when it violates system constraints. For example, instances on a node exceed its memory capacity.

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

```
1  def get_migrations(self, snapshot):
2    nodes = snapshot.nodes
3    no_nodes = len(nodes)
4    migration_percentage = float(CONF.consolidator.migration_percentage) / 100
5    no_inst = len(snapshot.instances_migrable)
6    no_inst_migrate = int(no_inst * migration_percentage)
7
8    # if no_inst_migrate == 0
9    # or no_nodes < 2, then
10   # return empty list.
11   # Cannot migrate.
12
13   migs = []
14   while no_inst_migrate > 0:
15     nodes_cpy = list(nodes) # copy nodes list
16
17     from_host = choose_host(nodes_cpy)
18     # choose_host code is skipped.
19     # The chosen node is randomly chosen
20     # taking into account that it has to host
21     # at least one instance.
22
23     inst_on_host = from_host.instances_migrable
24     no_inst_on_host = len(inst_on_host)
25
26     top_bound = min(no_inst_on_host, no_inst_migrate)
27     n = random.randint(1, top_bound)
28     no_inst_migrate -= n
29
30     instances = random.sample(inst_on_host, n)
31     nodes_cpy.remove(from_host) # do not choose same host
32     to_host = random.choice(nodes_cpy)
33     for i in instances:
34       migs.append(self.Migration(i, to_host))
35
36   return migs
```

Listing 5.4: Code for random algorithm

father and mother, and crossed them; now we choose only one chromosome and massively[20] 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 the 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. Selection algorithm plays its role when its time to decide which chromosomes to cross (in our case, mutate) to generate a new children to add to the new population (an implementation of tournament selection is provided in `nova.consolidator.ga.functions.TournamentSelection`).

fitness_function (Defaults to `nova.consolidator.ga.functions.NoNodesFitnessFunction`) Fitness function is the one thats establishes how much the chromosome fits the solution wanted (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 above what 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 `population_size` option. At each step the best `N` chromosomes (according to the fitness function used) will pass to the next epoch.

[20] We change the value of an high percentage of its genes.

```
1 class NoNodesFitnessFunction(FitnessFunction):
2   # The higher the less nodes are used:
3   # - no_nodes = 1: fitness = 1
4   # - no_nodes -> infinite: fitness -> 0
5
6   def get(self, chromosome):
7     return float(1) / len(set(chromosome))
```

Listing 5.5: Code for `NoNodesFitnessFunction`

There is another option which is `best` (defaults to *False*). After running some simulation, we discovered that most of the epochs run without improving the fitness of the best chromosome and so we spent time in generating useless children. To overcome this problem we revisited mutation. Mutation is applied changing a gene's value and maintaining the chromosomes validity. To change a gene value means moving an instance to another compute node. The other compute node, normally, is chosen randomly among suitable nodes[21]. When `best` is set to *True*, the other compute node is no more chosen randomly but as the best[22] among suitable 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 selection logic and mutation.

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

[21] Nodes that, hosting the machine, will not exceed their capacity in terms of vCPUs, memory and disk.

[22] The most busy compute node.

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

As well as genetic algorithm, holistic algorithm[23] is taken from the state of the art (see 3.2.2). In algorithm 4 we provide a high-level pseudo-code for holistic algorithm.

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

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
```

Chapter 6

Evaluation

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

[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.

with legacy networking, before the controller becomes a management bottleneck, 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].

[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`).

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 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.

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 container 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.

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).

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 containers. 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

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

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

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 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 inter-

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

[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`,

val 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. The consolidation interval used was thought to make consolidators highly 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 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

live-migration and resize operation. The bug is tagged as "invalid" because "[...] This is just beyond scope of the current fake driver [...]". However, we think that the lack of resize operations doesn't compromise simulation results. It's create and destroy operations which are the real building blocks of a cloud system.

- `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].

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

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

[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.

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 because of 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 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 consolidator considered 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 to influence mutation turning it into a non-random one. Mutation 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. Holistic algorithm does almost what "best" variant does when moving instances, but it is very far from the improvement given by it (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 terrible 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

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

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 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 (vCPUs, 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 (only 0.014 percent of difference).

Chapter 7

Conclusions and future works

Due to the two-topic nature of this thesis we split this chapter into two sections. Section 7.1 discusses about conclusions on results obtained from tests on aDock system (see section 6.1) and possible future work on the system itself. Section 7.2 discusses conclusions on results obtained from consolidation algorithms testing (see section 6.2) and future work in the field of virtual machine consolidation in OpenStack.

7.1 aDock

Tests conducted on aDock system confirmed our suppositions on aDock. It is possible for a simple developer or researcher to develop on its laptop and run simulations against an OpenStack system using aDock. Results obtained give reasonable starting times for the entire architecture. We can say that aDock is "lightweight". However we think that a comparison between aDock and one of the other options available at the state of the art (e.g. *Chef*. See subsection 3.3.2) would be a must. Direct comparison is necessary to understand if aDock is really better then its "competitors". To our detriment there is to say that, by construction, containers make aDock more lightweight than an architecture with hypervisor and virtual machines (see paragraph 4.2.2) #**TODO** "figura dei cioccolatai"??.

Future work on aDock is vast. For what concerns FakeStack, it could become a *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 total contrast with Docker philosophy and makes impossible for FakeStack to be "flexible". 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. In this case we should provide a templating language to make FakeStack automatically deploy an OpenStack architecture as given by the user, as *Chef* and *Puppet* already do. According to us and with the adequate support by the OpenStack community, FakeStack could become the equivalent, but Docker-powered, of *Chef-OpenStack* and *Puppet-OpenStack* in the OpenStack world.

For what concerns the templating language and automate deploying of an OpenStack system, Docker recently released tools for container orchestration[2] among which we can find *Compose*[3] which is "a way of defining and running multi-container distributed applications with Docker". We think that this functionality fits perfectly with what we need into FakeStack. Compose allows the user to create a `docker-compose.yml` file and start its newly defined system running `docker-compose up` and 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

[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/`

as a proof of concept[4]. Keystone (`key`) container depends on MySQL (`db`) and RabbitMQ (`rabbit`) containers, as well as 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 into 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 Nova services, such as the scheduler, the API and so on.

In modular case, every image maps to a single OpenStack service. Images provided should fit users needs to choose OpenStack's running code (as already said in non-functional requirement 1. See 4.2.2). For this reason we think that we could use DevStack again to install each single service. This choice would allow the user to choose GitHub repository URL and branch and to configure the service itself (see subsection 4.3.3). The user should create a different configuration file for each service (e.g. `keystone.conf` and `nova.conf`) containing the repository URL and branch used and the different installation phases of the service with their specific configuration. At this point, we could merge each different configuration file to a main `local.conf` file which specifies the different `ENABLED_SERVICES`, in this case only 1, for each of the images.

OpenStack configuration is not "hot-reloaded" at every modification, thus, implies 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.

For what concerns Oscard, we could give the possibility to user to run simulations with a composition of *all* possible operations that OpenStack allows

[4]Not all necessary services are listed. Ports are avoided. Images are supposed to be available.

```
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`

to perform. Only create, resize and destroy operations are supported up to now. Another point is to support aggregates of simulations. It happens that a user wants to run a group of simulations and extract averages of the aggregates already stored in Bifrost by Oscard. It was our case when we came to groups of 50 simulations, each block with a different consolidator. To calculate the numbers in table 6.4, we wrote a python script which extracted averages of aggregates from each group of simulations, knowing starting and ending simulation ID of each group. We suppose that a situation like this one could happen very often to users. Oscard should allow the user to give an unique label to a group of simulations and automatically extract the averages (also standard deviations would be good) of the aggregates which Oscard already calculates. Polyphemus should display those new data and represent someway the concept of *group* of simulations, of course.

7.2 Virtual Machine Consolidation in OpenStack

Tests run on different consolidators confirmed what we thought about OpenStack's virtual machine consolidation. This practice introduces high resource usage improvements with respect to "vanilla" OpenStack and so, it could lead to energy saving in the whole system. The best algorithm found, in fact, brought to about 14% increase in vCPUs usage; about 20% increase in RAM usage; about 1.5% increase in disk usage and a decrease of about 3 active nodes and so, of a 30% (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. When a nodes is inactive it can be powered off. If the time in which the node is inactive is too short, it could be a nonsense to

turn it off, because the system could need it while this is happening. It could be that some algorithms doesn't allow power off and so, they give a "fake" improvement. Resource usage increases, but the system cannot exploit this efficiency trait being impossible for it to turn a server off.

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).

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 type in percentage (e.g. create vs destroy operations) and their density through time. Up to now, in fact, we only supposed that operation density is not homogeneous introducing *NOP* operations and we introduced arbitrary operations weights.

During all simulations instances where supposed to run at a maximum workload. Their resource usage, in fact, is calculated directly from their flavor. It could be interesting to simulate different workloads on instances basing on the type of application they are running. 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 to now, our metrics doesn't involve number of migrations performed. Every live migration performed, in fact, has a cost in terms of energy and time **#TODO** ref to paper! download paper please. 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 wasn't and isn't in strong evolution. It seems that the community, in this moment, has to deal with greater problems, *Kilo* version of OpenStack, in fact, is to be released. However we understood and we used the power of `fake` mod-

ule and we want to fix its bugs and enhance it. During `nova-consolidator` service development, we had to fix (locally, up to now) a bug in live migration feature[5] and to accept the resize operation problem (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` option). This was necessary for us to simulate different architectures and limit in 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[6]. We think in the next future to open blueprints[7] both for `MStandardFakeDriver` and for `DynamicWLInstance` and to improve Nova's "fake" implementation[8]. It could be interesting to somehow simulate nodes' energy consumption in `FakeDriver` or directly out of OpenStack. Up to now, it would be a nonsense to extract nodes' energy consumption, given that our nodes are virtualized into Docker containers.

In section 6.1, we have 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 in the way to make weighers prefer stacking over spreading[9]. 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 perform only destroy and *NOP* operations and compare different consolidators in this perspective. It is useful to see consolidators in action starting from an empty system and

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

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

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

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

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

experimenting 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” left by them.

Appendices