Politecnico di Milano

Facoltà di Ingegneria dell'Informazione

Corso di Laurea Magistrale in Ingegneria Informatica

Dipartimento di Elettronica, Informazione e Bioingegneria

Towards Virtual Machine Consolidation in OpenStack

Advisor: Sam Jesus Alejandro Guinea Montalvo

Master thesis by:

Giacomo Bresciani	**matr. 804979**
Lorenzo Affetti	**matr. 799284**

Academic Year 2013-2014

dedica. . .

Ringraziamenti

Milano, 1 Aprile 2005

Affear

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

Chapter 2

OpenStack and 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 12) 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.

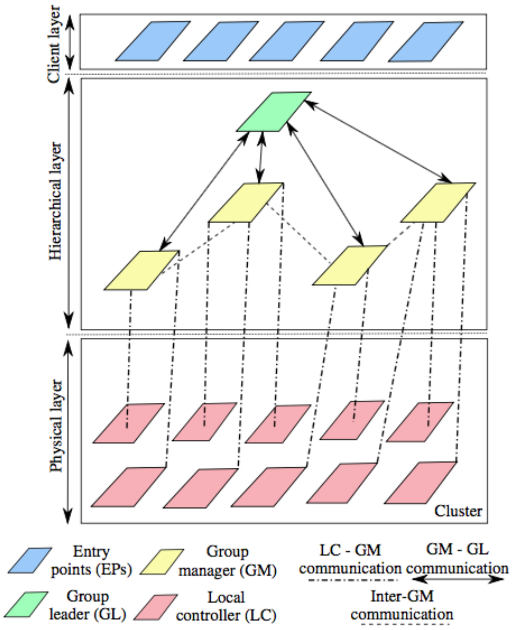


Figure 3.1: The Snooze architecture [?]

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 subproblems [**?**, 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`

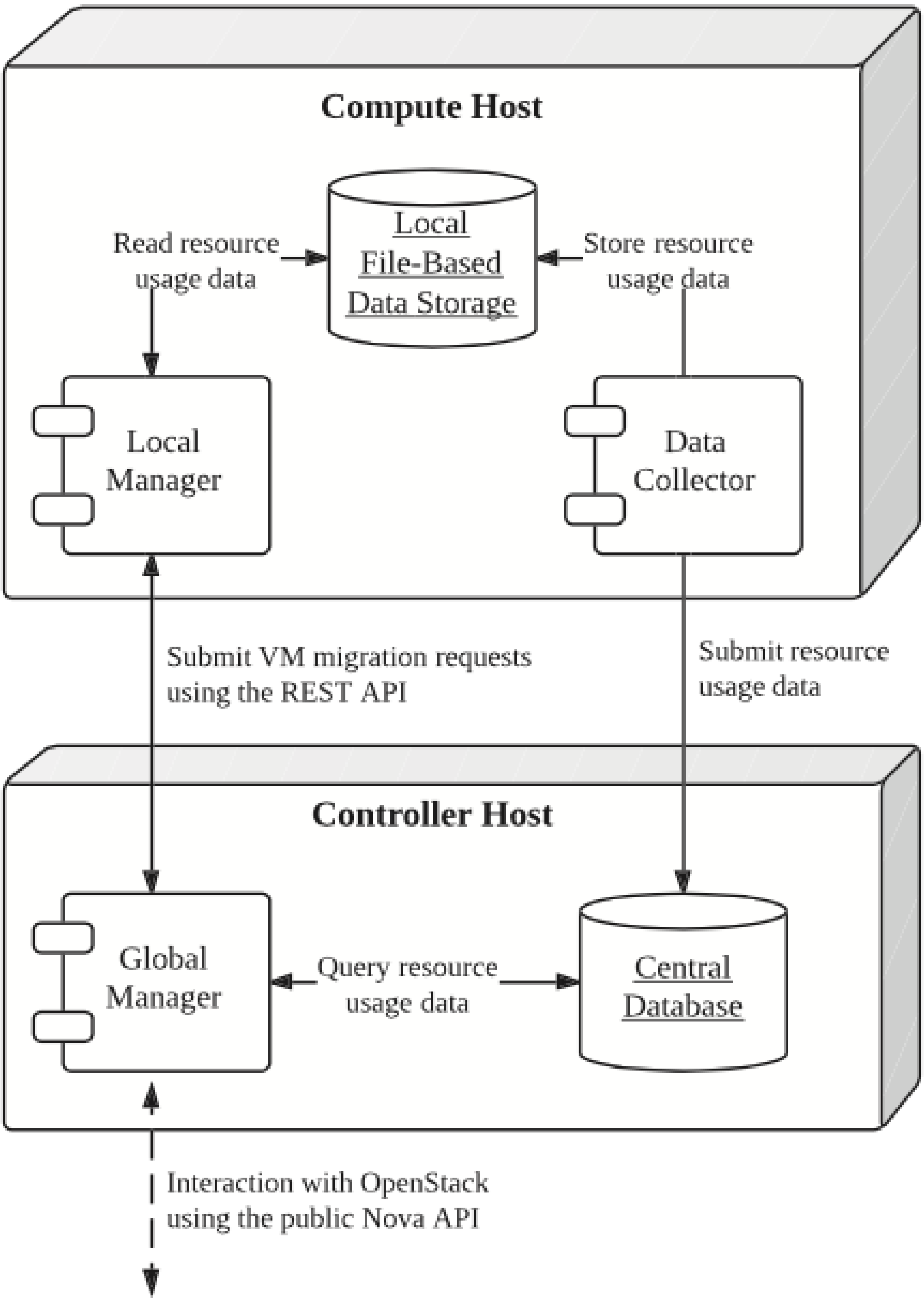

Compute Host
Local File-Based Data Storage
Read resource usage data
Store resource usage data
Local Manager
Data Collector
Submit VM migration requests using the REST API
Submit resource usage data
Controller Host
Global Manager
Query resource usage data
Central Database
Interaction with OpenStack using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

Figure 3.2: The OpenStack Neat architecture [?]

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

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 `chef-server` that orchestrates all the Chef processes, multiple `chef-client` found on all the servers, and the user workstation that communicates with the

[6] `www.chef.io`

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

[8] `supermarket.chef.io`

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 Compute” 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 Vagrantfile that configure a VirtualBox virtual machine that will host and All-in-One installation. Here is a part of it:

```
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',
```

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

```
      "#{File.dirname(__FILE__)}/.chef/encrypted_data_bag_secret")
  converge true
```

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 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 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 different results can be achieved:

[10] `www.puppetlabs.com`

[11] `forge.puppetlabs.com`

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

Controller node:

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

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
}
```

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 25, hosts a complete Operating System including application, dependency libraries, and, more important, the kernel; otherwise the Docker Engine runs as an isolated process in userspace on the host operating system and allows all the guest containers to share the kernel. Thus, it enjoys the resource isolation and allocation benefits of Virtual Machines but is much more 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... .

[13] www.docker.com

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.

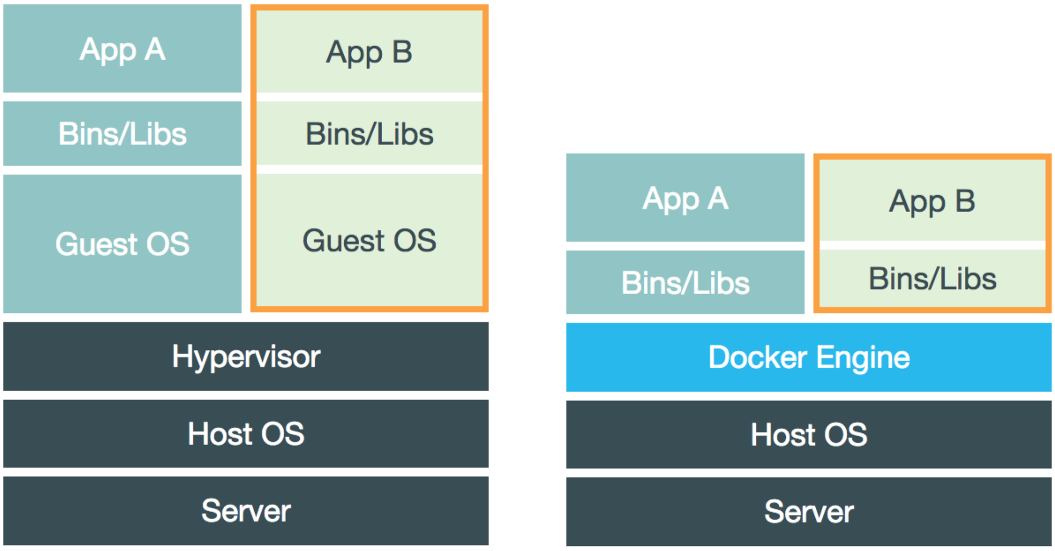


Figure 3.3: Hypervisor and Docker Engine

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.

[14] hub.docker.com

[15] github.com/ewindisch/dockenstack

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.

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 **??** we highlight the general architecture of aDock.

#TODO make aDock high-level architecture!

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/`

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 (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]

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

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

#TODO is the ref 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 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[4]. Configuring a service is as simple as adding few lines to `local.conf` file:

```
... # DevStack configurations

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

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

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

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

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.

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[5] 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*[6]. Both of them are Ubuntu-based Docker containers shipped with pre-installed software that satisfies most[7] 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

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

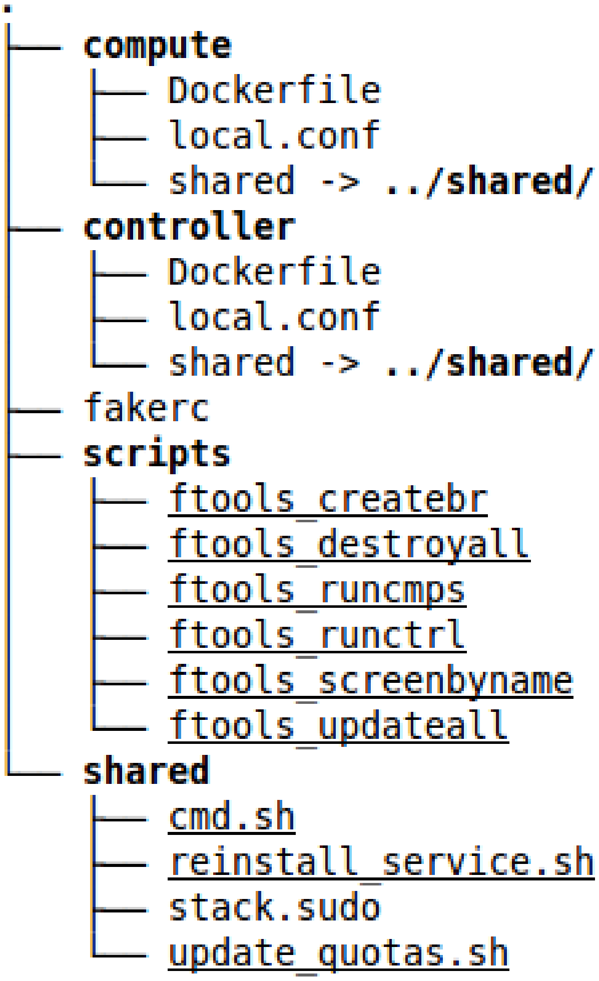
[6] In FakeStack, compute nodes, by default, are equipped with `nova.virt.fake.Fakedriver`. Thus, FakeStack compute nodes don't host a real hypervisor such as *Libvirt*. Virtual machines, in FakeStack, are mere python objects. This fact, doesn't influence user's choices. A user, in fact, can equip FakeStack's 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).

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

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.

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



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

Figure 4.1: FakeStack's file structure

As we can see in figure 4.3.1, nodes have two separated `Dockerfiles` (which makes them two different Docker containers), but they share a set of scripts contained in `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*[8] 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[9] to node's file-system avoiding rebuilding at each modification.

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

[9] `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[10]. 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-

[10] `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 on a new window in screen `running`. It is necessary to confirm the operation in the window created. Once confirmed, a new window, attached to one of the compute nodes started, will be opened.

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>`.

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**[11].

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`[12].

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

... # Other configuration options
```

Listing 4.1: Choose OpenStack's enabled services

```
... # Other configuration options

[[post-config|$NOVA_CONF]]
```

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

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

```
[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

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

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

DATABASE_TYPE=mysql

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

SERVICE_HOST=controller
MYSQL_HOST=controller
RABBIT_HOST=controller

```

```
NOVA_VNC_ENABLED=False
VIRT_DRIVER=fake

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

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

Listing 4.4: 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[13] 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.

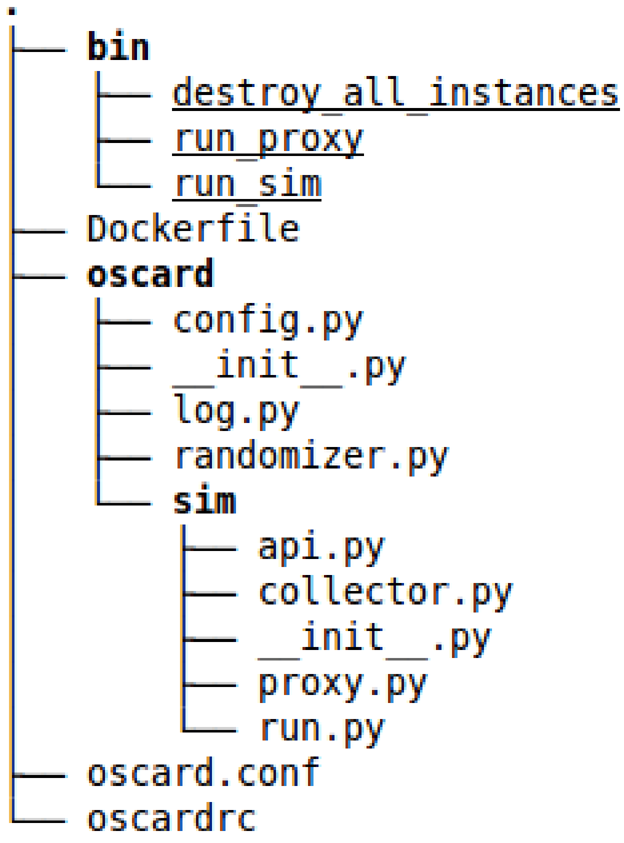
4.4.1 Modules

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).

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



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

Figure 4.2: 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[14]) 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[15] 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

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

[15] 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[16] 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[17] 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[18] 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`[19] 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[20]. Oscard, as FakeStack, has a source

[16] In `oscard.conf`: `no_t`

[17] Create Resize Destroy

[18] In `oscard.conf`: `<command-name>_w`

[19] Two of possible instance states in OpenStack. See `http://docs.openstack.org/developer/nova/devref/vmstates.html`

[20] 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

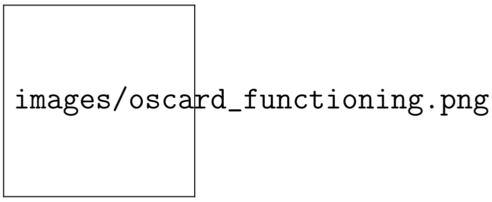


Figure 4.3: Workflow for a `create` operation

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.

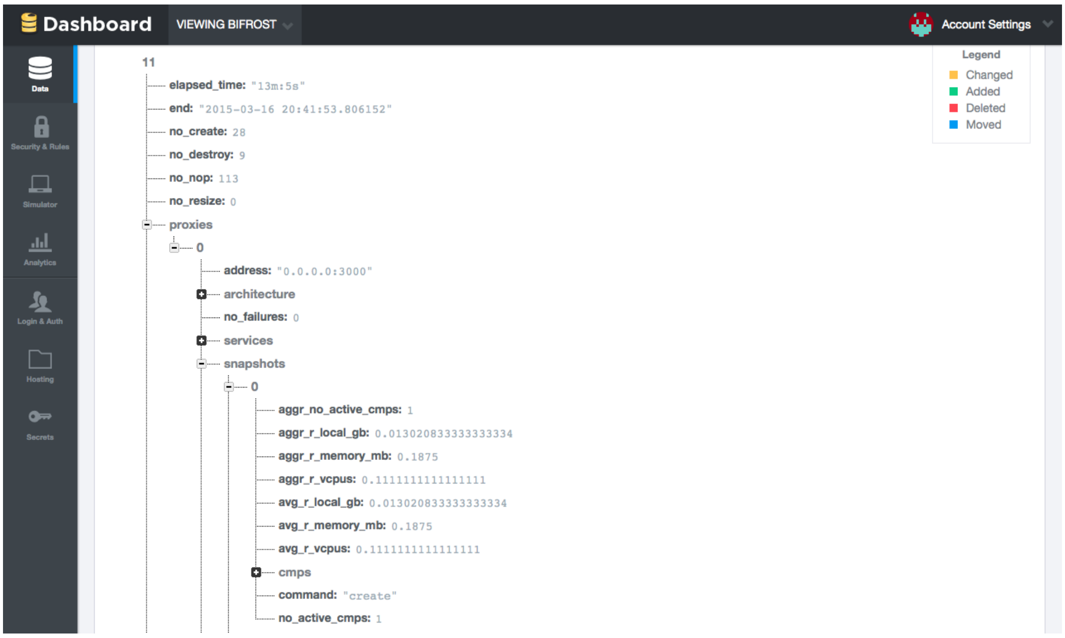
4.5.1 Bifrost

Bifrost[21] 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

[21] `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[22], so, our simulations *cannot* run concurrently[23].

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[24] operations on data.



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

Figure 4.4: Firebase Dashboard

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

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

[24] Create Read Update Delete

4.5.2 Polyphemus

Polyphems[25] is the Polymer[26]-powered view of aDock. It can be run in a Docker container or not[27]. 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[28] through line charts and percentage of resources usage[29] for each compute node through bar charts. A tool-bar representing data aggregates[30] 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.

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

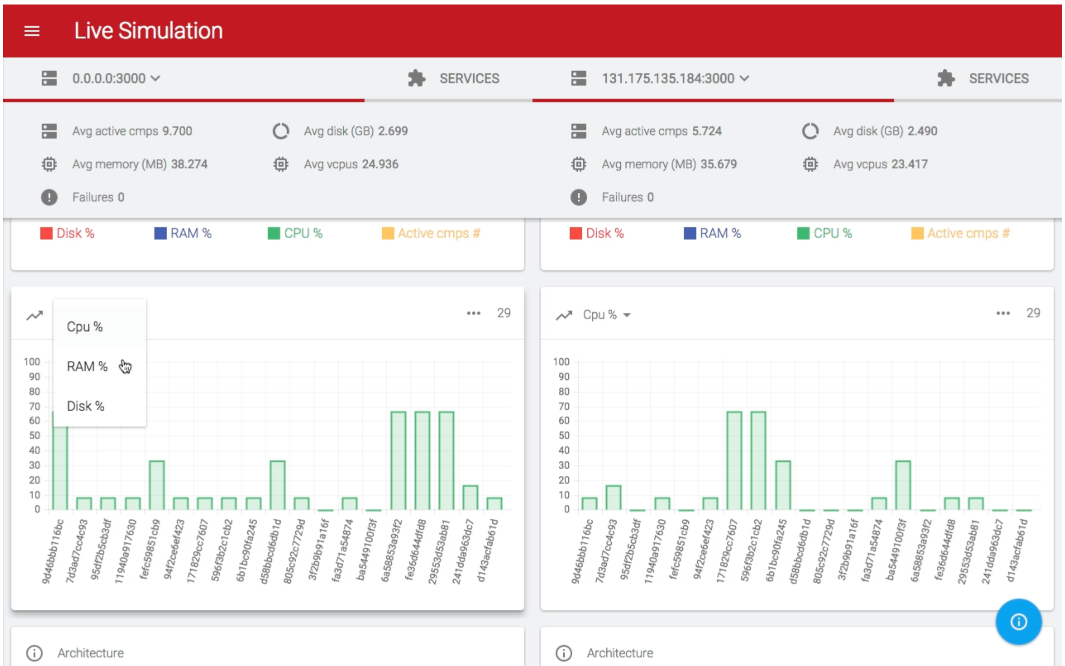
[26] `https://www.polymer-project.org/0.5/`

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

[28] 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).

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

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



Live Simulation

0.0.0.0:3000

Avg active cmps 9.700

Avg disk (GB) 2.699

Avg memory (MB) 38.274

Avg vcpus 24.936

Failures 0

131.175.135.184:3000

Avg active cmps 5.724

Avg disk (GB) 2.490

Avg memory (MB) 35.679

Avg vcpus 23.417

Failures 0

Figure 4.5: Screenshot of Polyphemus

4.6 aDock's Architecture

aDock is a modular system, where each component is run in its dockerized version[31]. 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[32] using a web browser on user's laptop.

[31] Apart from Bifrost.

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

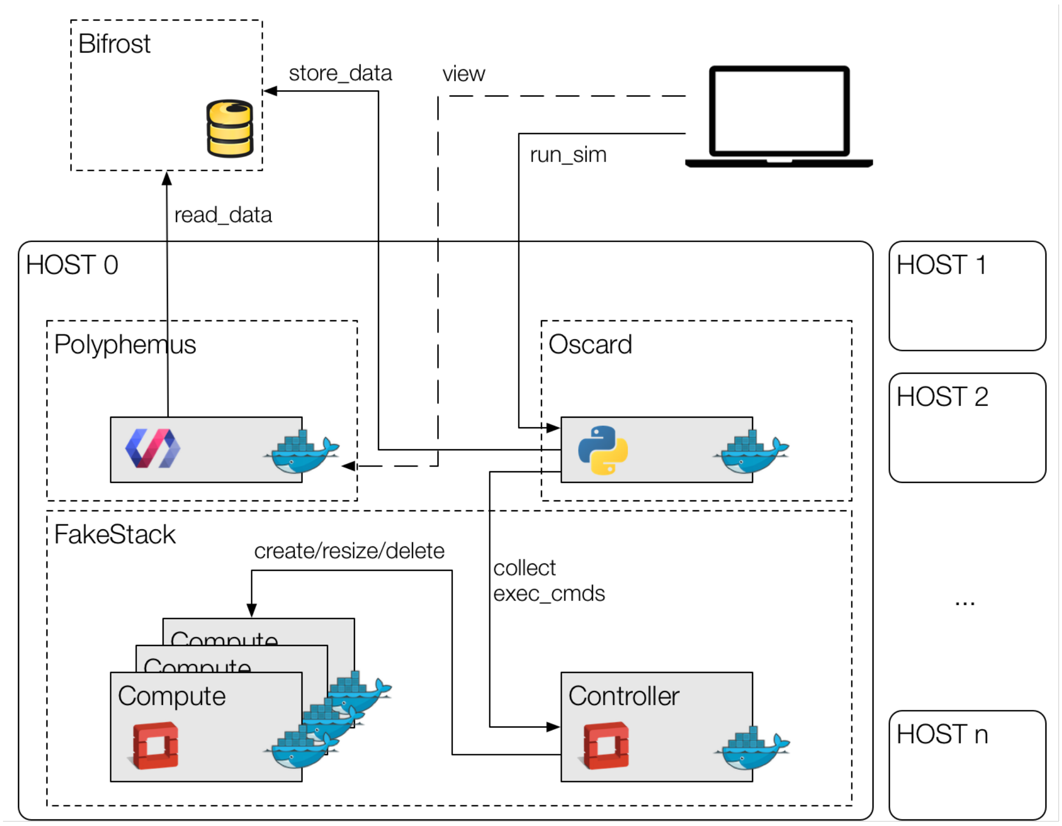


Figure 4.6: Screenshot of Polyphemus

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); the *manager*[4], which contains the real logic of the service and the *RPC*[5] *API*[6], which is used among OpenStack services to communicate[7].

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[8], `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 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[8]), which does nothing but defining a base class to be extended

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

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

[5] Remote Procedure Call

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

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

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

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[9] 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[10].

In detail, a `Snapshot` object offers active nodes (`nodes` attribute) and all, running, migratable[11] and not instances. Instances are

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

[10] 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.

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

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

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

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

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

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

```
class BaseConsolidator(object):

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

  # _transitive_closure method
  # implementation skipped

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

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

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

`nova.objects.instance.Instance`[12] objects; nodes are wrappers for `nova.objects.compute_node.ComputeNode`[13] objects, which add the possibility to get all, running, 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[14]. This algorithm was implemented for testing purpose and to see if even randomization could bring improvement in resource optimization, given that virtual machines are never moved in OpenStack[15].

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

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

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

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

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

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

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

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

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

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[8].

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[16].

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[17]. 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[18] two chromosomes, father and mother, and crossed them; now we choose only one chromosome and massively[19] mutate it.

The algorithm is configurable in all of its aspects:

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

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

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

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

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

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

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

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

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

    inst_on_host = from_host.instances_migrable
    no_inst_on_host = len(inst_on_host)

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

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

  return migs
```

Listing 5.4: Code for random algorithm

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.

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

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

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

Listing 5.5: Code for `NoNodesFitnessFunction`

compute node, normally, is chosen randomly among suitable nodes[20]. When `best` is set to *True*, the other compute node is no more chosen randomly but as the best[21] 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.

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

[21] 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[22] is taken from the state of the art (see 3.2.2). In algorithm 4 we provide a high-level pseudo-code for our genetic algorithm.

[22] 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


    return new_state
```

Chapter 6

Evaluation

Chapter 7

Conclusions and future works

Appendices