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

State of the art

2.1 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, VM placement, and 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 the 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 describe the state of the arts of Virtual Machine consolidation and of Cloud test environments.

2.2 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 Machine 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.

2.3 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 [18] and Puppet [19], 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 2.3.2) and Puppet (see section 2.3.3)

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.

2.3.1 Vagrant

Vagrant[1] 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 [2] 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:

```
1  box      = 'trusty64'
2  url      = 'http://files.vagrantup.com/precise32.box'
3  hostname = 'customtrustybox'
4  domain   = 'example.com'
5  ip       = '192.168.0.42'
6  ram      = '2048'
7
8  Vagrant::Config.run do |config|
9    config.vm.box = box
10   config.vm.box_url = url
11   config.vm.host_name = hostname + '.' + domain
12   config.vm.network :hostonly, ip
13
```

[1] www.vagrantup.com
[2] www.vagrantcloud.com

```
14    config.vm.customize [
15      'modifyvm', :id,
16      '--name', hostname,
17      '--memory', ram
18    ]
19  end
```

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.

2.3.2 Chef

Description Chef[3] 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)[4] 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

[3] `www.chef.io`

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

addition Chef offers a centralized hub, called Chef Supermarket[5], 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 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[6] there is a Vagrantfile that configure a VirtualBox virtual machine that will host and All-in-One installation. Here is a part of it:

[5] `supermarket.chef.io`
[6] `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
```

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...

2.3.3 Puppet

Description Similarly to Chef (described in section 2.3.2) Puppet[7] 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[8] 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 man-

[7] `www.puppetlabs.com`
[8] `forge.puppetlabs.com`

agement and others. The module is `puppetlabs-openstack`[9] 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:

Controller node:

```
1 node 'control.localdomain' {
2   include ::openstack::role::controller
3 }
```

Controller node:

```
1 node 'storage.localdomain' {
2   include ::openstack::role::storage
3 }
4
5 node 'network.localdomain' {
6   include ::openstack::role::network
7 }
8
9 node /compute[0-9]+.localdomain/ {
10  include ::openstack::role::compute
11 }
```

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.

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

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...

2.3.4 Docker

Docker[10] 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 2.3.4 on page 15, hosts a complete Operating System including application, dependency libraries, and, more important, the kernel; otherwise the Docker Engine runs

[10] `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[11] where you can upload both Dockerfile and pre-built container images to streamline the sharing process.

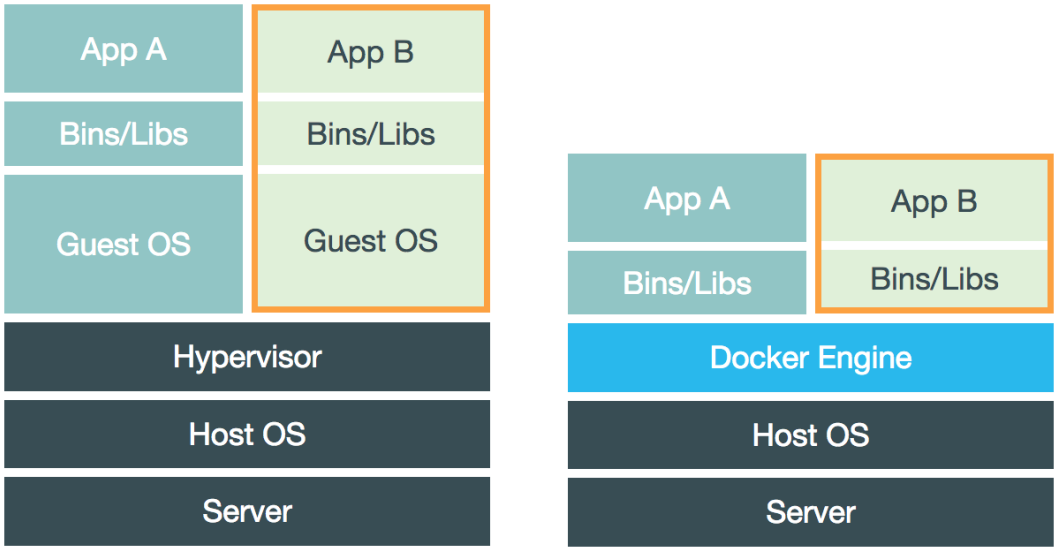

Figure 2.1: Hypervisor and Docker Engine

[11] `hub.docker.com`

2.3.5 Dockenstack

One of the first an attempts to create a cloud test environments based on OpenStack and Docker is a Dockenstack[12]. 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.

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.

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

Chapter 3

aDock

3.1 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.

3.2 Requirements

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

3.2.1 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 3.3) 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 3.4) 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 3.5).

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 3.5) 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!

3.2.2 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:

```
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

[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 3.4).

3.3 FakeStack

FakeStack 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*. Both of them are Ubuntu-based Docker containers shipped with pre-installed software that satisfies most[5] of OpenStack's services dependencies. Both controller and compute nodes are configurable by means of simple configuration files 3.3.3.
FakeStack provides a set of scripts to handle node startup, service updating on live nodes and other features 3.3.2.

3.3.1 Nodes

As already anticipated in requirement 3.2.2, 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.

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

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

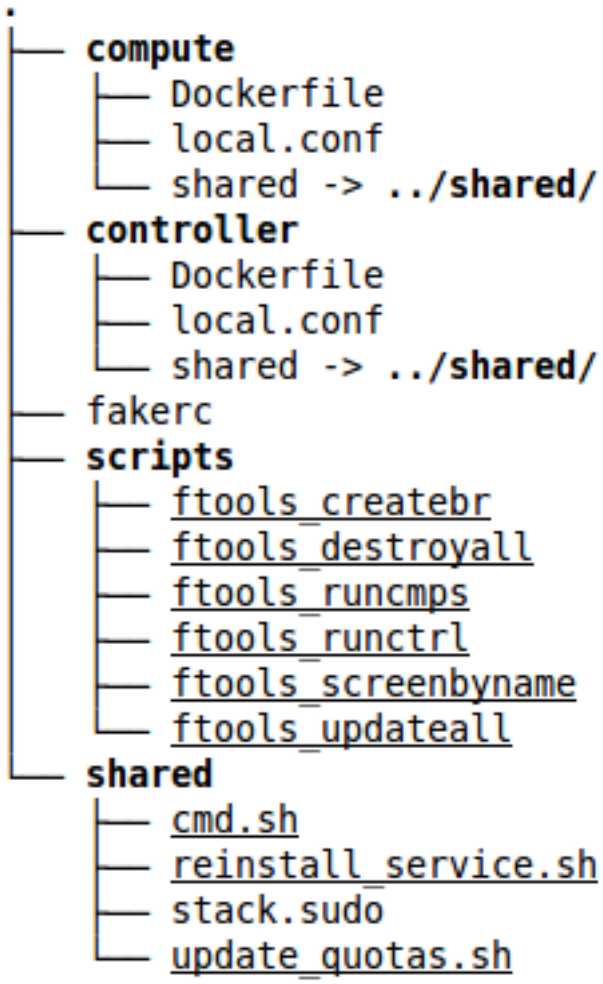

Figure 3.1: FakeStack's file structure

As we can see in figure 3.3.1, 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

update_quotas.sh

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[6] to node's file-system avoiding rebuilding at each modification.

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

3.3.2 Scripts

3.3.3 Configuration

using DevStack's syntax ...

[6] `local.conf` is a *Data Volume*. Basically it is a file whose modifications are visible at each container's run.

3.3.4 Example

3.4 Oscard

3.5 Other Components

Chapter 4

Nova Consolidator

Chapter 5

Evaluation

Chapter 6

Conclusions and future works

Appendices