A Private Cloud Implementation for MapReduce Applications

Marcos Salgueiro Balsa

I would like to thank my advisors **Dr. Patricia González Gómez**, **Dr. Tomás Fernández Pena** and **Dr. José Carlos Cabaleiro Domínguez** for their invaluable input and supervision efforts with this project.

Give a man a fish, and you'll feed him for a day.
Teach a man to fish, and you'll feed him for a lifetime.
Anne Isabella Thackeray Ritchie

[...] It takes these very simple-minded instructions – "Go fetch a number, add it to this number, put the result there, perceive if it's greater than this other number" – but executes them at a rate of, let's say, 1,000,000 per second. At 1,000,000 per second, the results appear to be magic.
Steven Paul Jobs

Summary

The history of computation has seen how the unending evolution of technology has promoted changes in its ways and means. Today, *tablets* and *smartphones*, quantitatively inferior managing and memorizing numbers, camp freely in a global market saturated with options. The tendency is clear: users will get to use more than one device to access the Internet and will want to have all their data synchronized and at hand all the time.

But that is only a part of the equation. At the other side of every service there lie a cluster of machines that must deal with an ever increasing traffic volume, while it maintains response delivery within outstanding delay times — low latency *may* have helped the infant Google rise above the competition. If we also added that the idea of surrounding every distributed application with energetic efficiency is a transcendental requisite and not simply a good practice, we would have the perfect setup for the proliferation of new paradigms like *Cloud Computing*. Cloud Computing — or just *the cloud* — is not intrinsically a new idea but an abstraction on an old one: *Virtualization*. The clouds' cornerstone is flexibility in providing clients with computational infrastructure.

Another technology that is constantly making headlines is *MapReduce*. If the cloud centers around easing infrastructure exploitation, MapReduce's core strength lies in its speeding up driving large masses of unstructured data. which, together, form an extraordinary computational tandem. The present text will put forth a solution that allows for drawing on the computational resources available, exploiting the cooperation of both technologies. Special emphasis has been placed in flexibility of access, being a web browser the only application required to use the service; in simplifying the virtual cluster configuration, by including a self-managed minimum deployment; and in transparency and extensibility, by freeing source code and documentation as open-source software, favoring its usage as starting point for larger installations.

Contents

List of Figures

Chapter 1

Abstract

Over the last years there has been a continuous increase in the amount of information generated with the Internet as the main driver. Furthermore, this information has reshaped from structured — and thus, susceptible to being expressed following a relational schema — to heterogeneous, which has kick-started the necessity to alter the way it is stored and transformed. As the figure 1.1 shows, those that were the undisputed back-end queens — mostly relational database systems — are seeing how their role is fading away due to their incapability to efficiently save unrelated heterogeneity.

As another related dimension, in the year 2000 many *.com* companies started upgrading their data centers to accommodate the inexorable demand peak that was going to follow. But it never came; and the bubble burst. What happened then was a general underutilization — only `10%` of Amazon's global computational resources were in use — that pushed the search for alternative means to export that surplus as a product. Amazon's own initiative unfolded in 2006 with the *AWS* (*Amazon Web Services*) appearance. AWS, among others, implements a public API for flexible on-demand infrastructure provisioning.

Since then, similar projects have proliferated generalizing how private clusters' unused computational capacity is to be serviced, trying to stay API-compatible with the AWS to facilitate interoperability and thus avoid their users' swapping to more flexible providers.

Meanwhile, Google was also in the search for new mechanisms to exploit, with high performance and securely, their own private infrastructure to evolve the capability of their services. MapReduce, as a way to massively execute thousands of transformations on input data, became a reality to thrust the generation of Google's humongous inverted index of the Internet [2]. Forthcoming contributions from Nutch's developers — by that time an Internet search engine prototype — to the MapReduce paradigm at *Yahoo!*, would traduce into the appearance of today's *de facto* standard in the field: Hadoop. Nowadays Hadoop is used in a myriad of backgrounds, ranging from travel booking sites to storing and servicing mobile data, e-commerce,

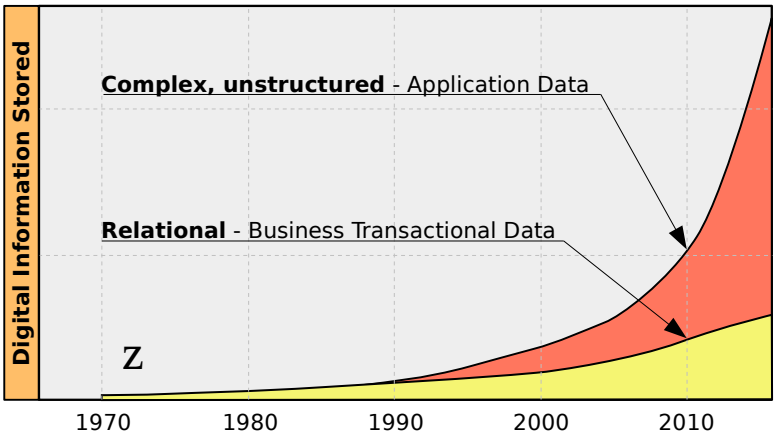

Figure 1.1: Demand in exponential growth. Source: *Cloudera Inc.*

image processing applications, searching for new forms of energy, business intelligence or web analytics.

So, by stacking a MapReduce implementation atop elastic infrastructure, an optimal exploitation of computational resources would be attainable rapidly expanding or shrinking them on-demand, while simultaneously reducing the overall energy consumption required to accomplish the processing (Figure 1.2).

1.1 Goals

The main goal this text pursues is to study the feasibility to develop a solution for a cloud to drive MapReduce applications, with no need to know the particular cloud structure and/or Hadoop configuration parameters.

In order to achieve such a simple execution model without compromising performance or applicability, a thorough analysis of different *IaaS Frameworks* will be carried out. Their features will be evaluated inside a virtual testing environment to finally narrow the selection to only one. Once an IaaS Framework had been chosen, the attention will be put towards choosing a MapReduce implementation to install over our virtual infrastructure.

Nonetheless, a mechanism to forward MapReduce execution requests will be devised and implemented trying to focus on simplicity and universal access. Yet, this transparency mustn't become an obstacle in exploiting the application or in fetching processed results. Privacy and security in communications and storage will be conveniently defined; we shan't forget it will be developed as a scaled-down model which could be infinitely scaled out.

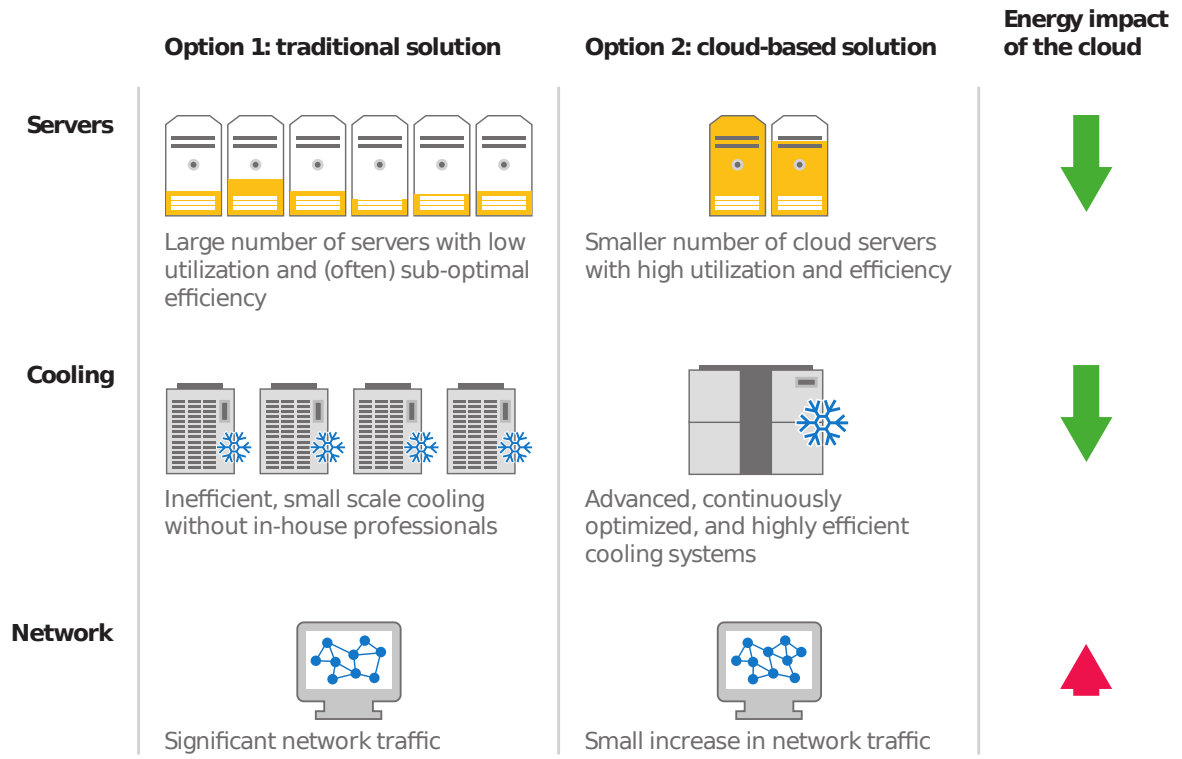

Figure 1.2: Energy savings. Source: [1]

1.2 Arrangement of the Document

The contents within this document are distributed as stated next. This first chapter introduces the development guidelines in the abstract. Chapter 2 puts the reader closer to the fundamental Cloud Computing concepts — like general architecture or virtualization —, along with those from the MapReduce paradigm. Chapter 3 describes an empirical evaluation of four private IaaS Cloud frameworks. Chapter 4 explores OpenStack Folsom's modular structure and particular inner workings. Likewise, chapter 5 unveils Hadoop's peculiarities as a MapReduce implementation.

The subsequent chapters center on detailing the project from diverse vantage points. Chapter 6 contains a series of design decisions and their accompanying UML diagrams. Chapter 7 gathers an analysis on performance in a real testing cluster. Chapter 8 analyzes related research projects highlighting how they compare to this solution. Finally, the main contributions of this project are discussed in addition to proposing future improvements to the implementation.

Two annexes have also been included. Annex A guides the reader throughout a quick single node installation and its operating maintenance. Annex B covers the definition of some of the concepts and technologies referred to in this text.

Chapter 2

Background

This second chapter tries to acquaint the reader with the key concepts that define Cloud Computing as well as the MapReduce archetype. Later chapters will elaborate on top of them.

2.1 Cloud Computing

As it has already been discussed, Cloud Computing is, in essence, a distributed computational model that attempts to ease on-demand consumption of computational infrastructure, by exporting it as virtual resources, platforms or services. However it may seem, Cloud Computing is no new technology; it introduces a new way to exploit idle computing capacity. What it intends is to make orchestration of enormous data centers more flexible, so as to let a user start or destroy virtual machines as required — Infrastructure as a Service (*IaaS*) —, leverage a testing environment over a particular operating system or software platform — Platform as a Service (*PaaS*) — or use a specific service like remote backup — Software as a Service (*SaaS*). Figure 2.1 shows the corresponding high level layer diagram of a generic cloud.

Different IaaS frameworks will cover the functionality that is required to drive the cloud-defining *physical* infrastructure. Nonetheless, an effort to analyze, design, configure, install and maintain the intended service will be needed, bearing in mind that the degree of elaboration grows from IaaS services to SaaS ones. In effect, PaaS and SaaS layers are stacked supported by those immediately under (figure 2.1) — as happens with software in general which is implemented over a particular platform which, in turn, is also build upon a physical layer. Every Cloud framework focuses on giving the option to configure a stable environment in which to run virtual machines whose computing capabilities are defined by four variables: Virtual CPU count, virtual RAM, virtual persistent memory and virtual networking devices. Such an environment makes it possible to deploy virtual clusters upon which to install platforms or services to be subsequently consumed by users, building up the software layers that conform PaaS and

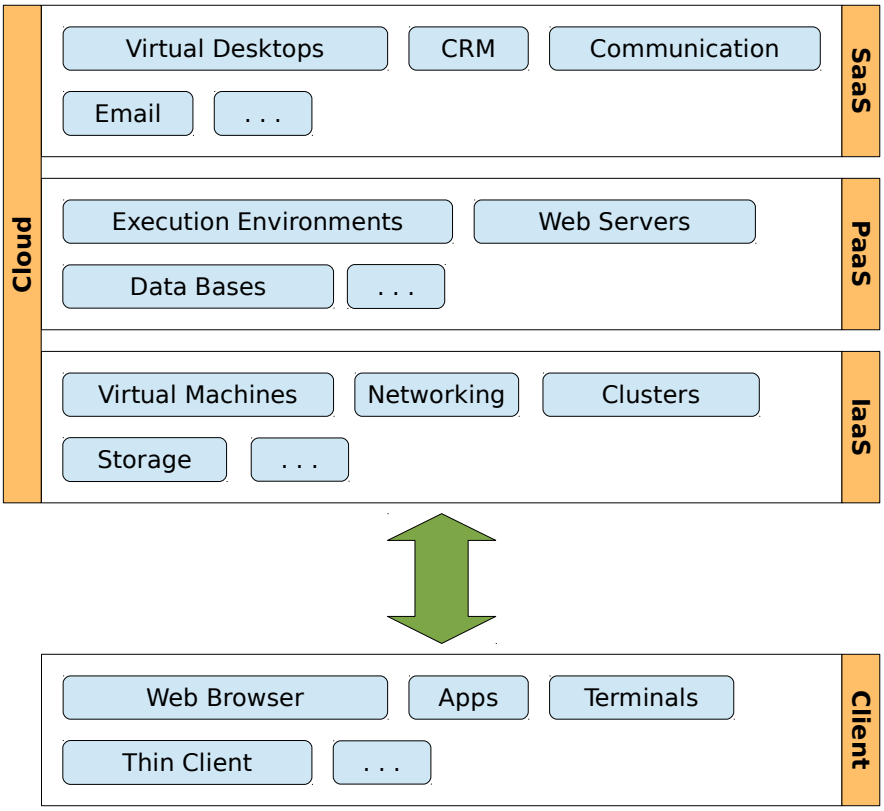

Figure 2.1: Software layers in a particular cloud deployment

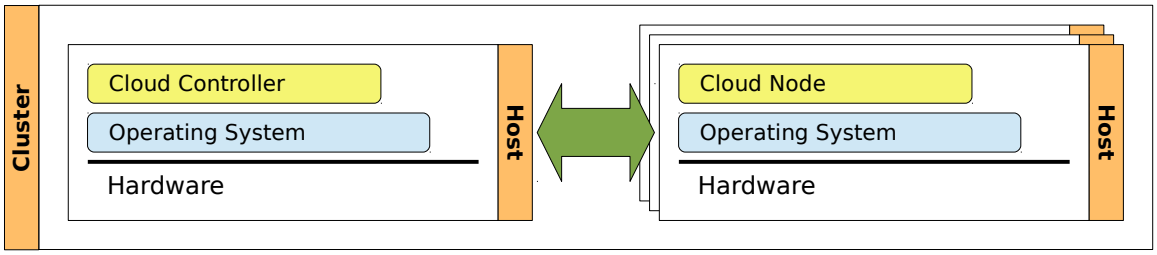

Figure 2.2: Cloud Controller and Cloud Node

SaaS paradigms respectively.

No less important features like access control, execution permissions, quota or persistent or safe storage will also be present in every framework.

2.1.1 Architecture

Figure 2.1 showed possible layers that could be found in a cloud deployment. Depending on the implemented layers, the particular framework and the role played by the cluster node, there will appear different particular modules making it possible to consume the configured services. These modules may be though of as cloud subsystems that connect each one of the parts that are required to execute virtual machines. Those virtual machines' capabilities are defined by the four variables previously discussed — VCPUS, RAM, HDD and networking. As there is no methodology guiding how those subsystems should be in terms of size and responsibility, each framework makes its own modular partition regarding infrastructure management.

Setting modularity apart, one common feature among different clouds is the separation of responsibility in two main roles: *Cloud Controller* and *Cloud Node*. Figure 2.2 shows a generic cloud deployment in a cluster with both roles defined. The guidelines followed for having this two roles lay close to the *Master-Slave* architecture approach. In those, in general, there's a set of computers labeled as coordinators which are expected to control execution, and another set made up with those machines that are to carry out the actual processing.

In a cluster, host computers or cluster nodes — labeled as Cloud Controllers or Cloud Nodes — cooperate in a time synchronized fashion with *NTP* (*Network Time Protocol*) and communicate via *message-passing* supported by asynchronous queues. To store services' metadata and status they typically draw upon a *DBMS* (*Data Base Management System*) implementation, which is regularly kept running in a dedicated cluster node, or sharded (distributed) among the members of a database federation.

Although there is no practical restriction to configure both Cloud Controller and Cloud Node within a single machine in a cluster, this approach should be limited to development or

prototyping environments due to the considerable impact in performance that it would imply.

Cloud Controller

The fundamental task of a Cloud Controller is to maintain all of the cloud's constituent modules working together coordinating their cooperation. It is a Controller's duty to:

- Control authentication and authorization.
- Recount available infrastructure resources.
- Manage quota.
- Keep track of usage balance.
- Maintain an inventory of users and projects.
- Expose an API for service consumption.
- Monitor the cloud in real time.

Being an essential part of a cloud as it is, the Controller node is usually replicated in physically distinct computers. Figure 2.3 shows a Cloud Controller's architecture from a high level perspective.

As a general rule, clients will interact with the cloud through a Web Service API — *RESTful* APIs (*REpresentational State Transfer*) mostly. Those APIs vary slightly from company to company, as usual, which forces clients to be partially coupled to the cloud implementation they intend to use. That is why there has been an increasing trend for unifying and standardizing those APIs in order to guarantee compatibility inter-framework. Of special mention is the cloud standard proposed by the *Open Grid Forum*: *OCCI* (*Open Cloud Computing Interface* [6]).

Cloud's modules support its functional needs. Each one of them will have a well-defined responsibility, and so, there will appear networking-centered modules, access and security control modules, storage modules, etc. Many of them existed before the advent of Cloud Computing, but they worked only locally to a cluster or organization. Inter-module communication is handled by an asynchronous message queue that guarantees an efficient broadcasting system off the Cloud Controller. To store and expose configuration data to the cluster in a single place while managing concurrent requests to update these data, every IaaS Framework evaluated resorts to a DBMS.

Hardware requirements for the cluster nodes vary from each particular framework implementation and the *QoS* expected, but they generally require something around 10 GB of RAM, quad core CPU, Gigabit Ethernet and one TB of storage to get started.

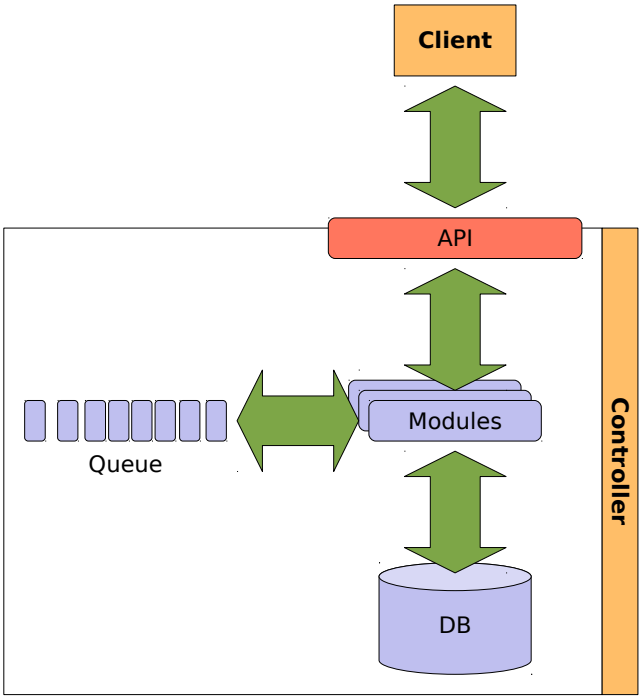

Figure 2.3: Cloud Controller in detail

Cloud Node

If the Cloud Controller is entrusted the cloud's correct functioning acting like glue for its parts, the actual task processing is performed in the Cloud Nodes using the VCPU, VRAM, VHDD that are going to be borrowed from the corresponding CPU, RAM and HDD from the real nodes of the cluster.

Cloud Nodes may be heterogeneous according to their hardware characteristics. They will configure a resource set that, seen from outside of the cluster, will appear to be a homogeneous whole where the summation of capacities of every participating node is the cloud's dimension. Furthermore, this homogeneous space could be provisioned, as discussed, on-demand. It is the Cloud Controller's responsibility to single out the optimal distribution of virtual servers throughout the cluster, attending to the physical aspects of both the virtual machine and the computer in which it will run.

The most important subsystem in a Cloud Controller is the *hypervisor* or *VMM* (*Virtual Machine Monitor*). The hypervisor is responsible for making possible the execution of virtual servers — or virtual instances — by creating the virtual architecture needed and the *virtual execution domain* that will be managed with the help of the kernel of the operating system. To generate this architecture there fundamentally exist three techniques: *Emulation*, *Paravirtualization* and *Hardware Virtualization* — or *Full Virtualization*. Different hypervisors support

them in different degree, and most will only cover one of them.

2.1.2 Virtualization Techniques

What follows is a brief review of the main methods to create virtual infrastructure.

Emulation

Emulation is the most general virtualization method, in a sense that it does not call for anything special be present in the underlying hardware. However, it also carries the highest penalization in terms of performance. With emulation, every structure sustaining the virtual machine operation is created as a functional software copy of its hardware counterpart; i.e., every machine instruction to be executed in the virtual hardware must be run software-wise first, and then be translated on the fly into another machine instruction runnable in the physical domain — the cluster node. The interpreter implementation and the divergence between emulated and real hardware will directly impact the translation overhead. This fact hinders the emulation from being widely employed in performance-critical deployments. Nonetheless, thanks to its operating flexibility, it's generally used as a mechanism to support legacy systems or as a development platform. Besides, the kernel in the guest operating system — the kernel of the virtual machine operating system — needs no alteration whatsoever, and the cluster node kernel need only load a module.

Hardware Virtualization

Hardware Virtualization, on the contrary, allows host's processes to run directly atop the physical hardware layer with no interpretation. Logically, this provides a considerable speedup from emulation, though it imposes a special treatment to be given to virtual processes. Regarding CPUs, both AMD's and Intel's support virtual process execution — which is the capacity to run processes belonging to the virtual domain with little performance overhead — as far as convenient hardware extensions are present (*SVM* and *VT-x* respectively [7]). Just as happened with emulation, an unaltered host kernel may be used. This fact is of relative importance as if wasn't so it would limit the myriad of OSs that could be run as guests. Lastly, it should be pointed out that the hardware architecture is exposed to the VM as it is, i.e. with no software middleware or transformation.

Paravirtualization

Paravirtualization uses a different approach. To begin with, it is indispensable that the guest kernel be modified to make it capable of interacting with a paravirtualized environment. When

the guest runs, the hypervisor will separate the regions of instructions that have to be executed in kernel mode in the CPU, from those in user mode which will be executed as regular host instructions. Subsequently, the hypervisor will manage an on-contract execution between host and guest allowing the latter to run kernel mode sections as if they belonged to the real execution domain — as if they were instructions running in the host, not in the guest — with almost no performance penalty. Paravirtualization, in turn, does not require an special hardware extension be present in the CPU.

2.1.3 Cloud IaaS frameworks

Cloud IaaS frameworks are those software systems managing the abstraction of complexity associated with on-demand provisioning and administering failure-prone generic infrastructure. In spite of being almost all of them open-sourced — which fosters reusability and collaboration —, they have evolved in different contexts. This fact has raised the condition of lacking interoperability among them, maturing non-standard APIs; though today those divergences are fading away. These frameworks and APIs are product of the efforts to improve and ease the control of the underlying particular clusters of machines on which they germinated. Thus, it is no surprising their advances had originated in parallel with the infrastructure they drove, leaving compatibility as a secondary issue.

Slowly but steadily these managing systems became larger in reach and responsibility boosted by an increasing interest in the sector. It happened that software and systems engineering evolved them into more abstract software packages capable of managing more different physical architectures. AWS appearance finished forging the latent standardization need, and as of today, most frameworks try to define APIs close to Amazon's and OCCI's [6].

2.2 MapReduce Paradigm

The origin of the paradigm centers around a publication by two Google employees [2]. In this paper they explained a method implementation devised to abstract the common parts present in distributed computing that rendered simple but large problems much more complex to solve when paralleling their own code on massive clusters.

A concise definition states that MapReduce is "*a data processing model and execution environment that runs on large clusters of commodity computers*" [8].

$$\textbf{map} : (\alpha \rightarrow \beta) \rightarrow \alpha\, list \rightarrow \beta\, list$$
$$\textbf{map}\ (\textbf{pow}\ 2)\ [1,2,3] \Rightarrow [1,4,9]$$

Figure 2.4: Map function example (functional version)

2.2.1 Programming Model

The MapReduce programming model requires the developer express problems as a partition of two well-defined pieces. A first part deals with the reading of input data and with producing a set of intermediate results that will be scattered over the cluster nodes. These intermediate transformations will be grouped according to an intermediate key value. A second phase begins with that grouping of intermediate results, and concludes when every *reduce* operation on the groupings succeeds. Seen from another vantage point, the first phase corresponds, broadly speaking, to the behavior of the functional *map* and the second to the functional *fold*.

In terms of the MapReduce model, these functional paradigm concepts have given rise to *map* and *reduce* functions. Both map and reduce have to be supplied by the developer, which may require breaking down the original problem in a different manner in order to accommodate to the model's execution mechanics. As counterpart, the MapReduce model will deal with parallelizing the computation, distributing input data across the cluster, handling exceptions that could raise during execution and recovering output results; everything transparent to the engineer.

Map Function

The typical functional map takes any function F and a list of elements L or, in general, any recursive data structure, to return a list resulting from applying F to each element of L. Figure 2.4 shows its signature and an example.

In its MapReduce realization, the map function receives a tuple as input and produces another tuple *(key, value)* as intermediate output. It is the MapReduce library's responsibility to feed the map function by transforming the data contained in input files into *(key, value)* pairs. Then, after the mapping has been completed, it deals with grouping those intermediate tuples by key before passing them in as input to the reduce function. Input and output data types correspond to those shown in the function signature in figure 2.5.

Reduce function

The typical functional fold expects any function G, a list L, or generally any type of recursive data type, and any initial element I, subtype of L's elements. Fold returns the value in I

$$\begin{array}{|l|}\hline \textbf{map} : (k1, v1) \rightarrow (k2, v2)\, list \\ \textbf{k} : clave \\ \textbf{v} : valor \\ (\textbf{kn}, \textbf{vn}) : par\ (clave, valor)\ en\ un\ dominio\ n \\ \hline \end{array}$$

Figure 2.5: Map function signature (MapReduce version)

$$\begin{array}{|l|}\hline \textbf{fold} : (\alpha \rightarrow \beta \rightarrow \alpha) \rightarrow \alpha \rightarrow \beta\, list \rightarrow \alpha \\ \textbf{fold}\ (+)\ 0\ [1, 2, 3] \Rightarrow 6 \\ \hline \end{array}$$

Figure 2.6: Fold function example

resulting from building up the intermediate values generated after applying G to each element of L. Figure 2.6 presents fold's signature as well as an example.

Contrary to map, reduce expects the intermediate groups as input to produce a smaller set of values for each group as output, because reduce will iteratively *fold* the groupings into values. Those reduced intermediate values will be passed in again to the reduce function if more values with the same key appeared as input. Reduce's signature is shown on figure 2.7. Just as happens with map, MapReduce handles the transmission of intermediate results out from map into reduce. The model also describes the possibility to define a *Combiner* function that would act after map, partially reducing the values with the same key to lower network traffic — the combiner usually runs in the same machine as the map.

A word counter in MapReduce

As an example, figure 2.8 shows the pseudocode of a MapReduce application to count the number of words in a document set.

In a Wordcount execution flow the following is going to happen: map is going to be presented a set of names containing all of the documents in plain text whose words will be counted. Map will subsequently iterate over each document in the set emitting the tuple *(<word>, "1")* for

$$\begin{array}{|l|}\hline \textbf{reduce} : (k2, v2\, list) \rightarrow v2\, list \\ \textbf{k} : clave \\ \textbf{v} : valor \\ (\textbf{kn}, \textbf{vn}) : par\ (clave, valor)\ en\ un\ dominio\ n \\ \hline \end{array}$$

Figure 2.7: Reduce function signature

```
Map (String key, String value):
// key:  document name
// value:  document contents
for each word w in value:
EmitIntermediate (w, "1");

Reduce (String key, Iterator values):
// key:  a word
// values:  an Iterable over intermediate counts of the word key
int result = 0;
for each v in values:
Emit (AsString (result));
```

Figure 2.8: MapReduce wordcount pseudocode. Source: [2]

each word found. Thus, an explosion of intermediate pairs will be generated as output of map, which will be distributed over the network and progressively folded in the reduce phase. Reduce is going to be input every pair generated by map but under a different form. Reduce will accept the pair *(<word>, list("1"))* on each invocation. The list of *"1"*s, or generically, an `Iterable` over *"1"*s, will contain as many elements as instances of the word *<word>* there were in the document set — supposing the map phase were over before starting the reduce phase and that every word *<word>* were submitted to the same reducer (a cluster node executing the reduce function) in the cluster.

Once the flow had been completed, MapReduce would return a listing with every word in the documents and the number of times it appeared.

2.2.2 Applicability of the Model

The myriad of problems that could be expressed following the MapReduce programming paradigm is clearly reflected in [2], a subset of them being:

Distributed grep: Map emits every line matching the regular expression. Reduce only forwards its input to its output acting as the identity function.

Count of URL access frequency: Like wordcount, but with the different URLs as input to map.

Reverse web-link graph: For each URL contained in a web document, map generates the pair *(<target_ URL>, <source_ URL>)*. Reduce will emit the pair *(target, list(source))*.

Inverted index: Map parses each document and emits a series of tuples in the form *(<word>, <document_id>)*. All of them are passed as input to reduce that will generate the sequence of pairs *(<word>, list(document_id))*.

2.2.3 Processing Model

Besides defining the structure that the applications willing to leverage MapReduce capabilities will have to follow — so that they need not code their own distribution mechanisms —, with [2] an implementation of the model was introduced which allowed Google to stay protocol, architecture and system agnostic while keeping their commodity clusters on full utilization. This agnosticism allows for deploying vendor-lock free distributed systems.

The MapReduce model works by receiving self-contained processing requests called *job*s. Each job is a *partition* of smaller duties called *task*s. A job won't be completed until no task is pending for finishing execution. The processing model main intent is to distribute the tasks throughout the cluster in a way that reduced job latency. In general, task processing on each phase is done in parallel and phases execute in sequence; yet, it is not technically needed for reduce to wait until map is complete.

Figure 2.9 shows a summary of a typical execution flow. It is interesting enough to deepen in its details as many other MapReduce implementations will present similar approaches.

1. MapReduce divides input files in M parts, the size of which is controlled with a parameter, and distributes as many copies of the MapReduce user algorithm as nodes participate in the computation.

2. From this moment onward, each program copy resides in a cluster node. A random copy is chosen among them and is labeled as the *Master Replica*, effectively assigning the *Master Role* to the node holding the replica; every other node in the cluster is designated with the *Worker Role*. Those worker nodes will receive the actual MapReduce tasks and their execution will be driven by the master node. There will be M map tasks and R reduce tasks.

3. Workers assigned map tasks read their corresponding portions of the input files and parse the contents, generating tuples *(key, value)* that will be submitted to the map function for processing. Map outputs are stored in memory as cache.

4. Periodically, those pairs in memory are *spilt* to a local disk and partitioned into R regions. Their path on disk is then sent back to the master, responsible for forwarding these paths to *reduce workers*.

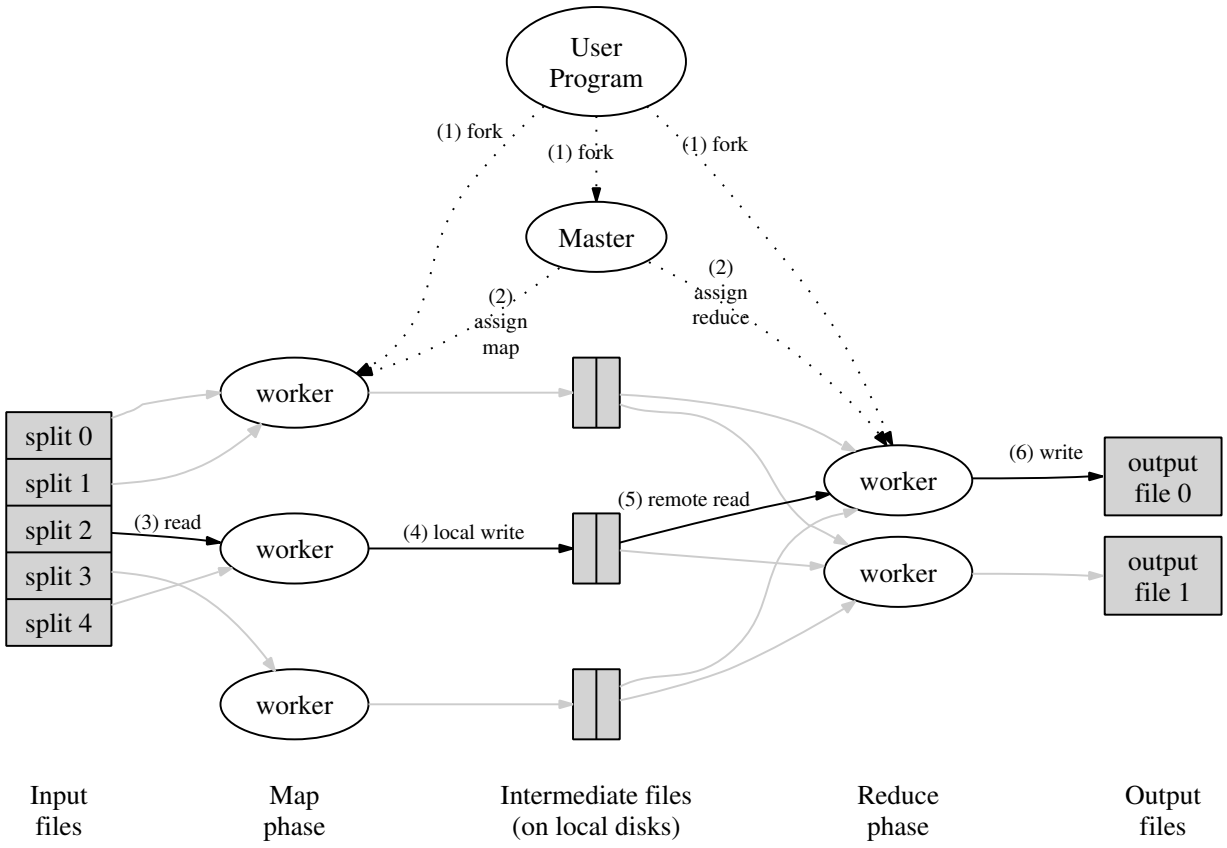

Figure 2.9: MapReduce execution diagram. Source: [2]

5. Now, when a reducer is notified that it should start processing, the path to the data to be reduced is send along and the reducer will fetch it directly from the mapper via *RPC* (*Remote Procedure Call*). Before actually invoking the reduce function, the node itself will sort the intermediate pairs by key.

6. Lastly, the reducer iterates over the key-sorted pairs submitting to the user-defined reduce function the key and the Iterable of values associated with the key. The output of the reduce function is appended to a file stored over the distributed file system.

When every map and reduce tasks had succeeded, the partitioned output space — the file set within each partition — would be returned back to the client application that had made the MapReduce invocation.

This processing model is abstract enough as to be employed to the resolution of very large problems running on huge clusters.

2.2.4 Fault Tolerance

The idea of providing an environment to execute jobs long enough to require large sets of computing machines to keep the latency within reasonable timings, calls for the definition of a policy able to assure a degree of tolerance to failure. If unattended, those failures would lead to errors; some would cause finished tasks to get lost, others would put intermediate data offline. Consequently, if no measures were taken to prevent or deal with failure, job throughput would humble as some would have to be rescheduled all along.

The MapReduce model describes a policy foreseeing a series of issues within an execution flow, and duly implements a series of actions to prevent them.

Worker Failure

The least taxing of the problems. To control that every worker is up, the master node polls them periodically. If a worker did not reply to pings repeatedly, it would be marked as *failed*.

A worker marked failed will neither be scheduled new tasks nor will be remotely accessed by reducers to load intermediate map results that it may had; a fact that could prevent the workflow from succeeding. If so were the case, the access to these data would be resolved by the master labeling the results of the failing tasks as *idle*, so that they could be rescheduled at a later time to store the results in an active worker.

Master Failure

Failure of a master node is more troublesome. The proposed approach consists in making the master periodically create a snapshot from which to restore to a previous state if it went

unexpectedly down. It is a harder problem than a worker failure mainly because there can only be one master per cluster, and the time it would take another node to take over the master role would leave the scheduling pipeline stalled. The master being in a single machine has, nonetheless, the benefit of lowering the probability of failure — this is precisely why the Google MapReduce paper [2] had put forward that the entire job be canceled. Still, as it is no good design decision to leave a *single point of failure*, subsequent MapReduce implementations have proposed to replicate the master in other nodes in the same cluster.

2.2.5 Aditional Characteristics

What follows is a summary of additional features of the original MapReduce implementation.

Locality

The typical bottleneck in modern distributed deployments is network bandwidth. In MapReduce executions, the information flows into the cluster from an external client. As already discussed, each node in a MapReduce cluster holds a certain amount of the input data and shares its processing capacity to be used for particular MapReduce tasks over those data. Each stage in the MapReduce executing pipeline requires a lot of traffic to be handled by the network which would reduce throughput if no channel wide enough were deployed nor a locality exploiting strategy were implemented.

In fact, MapReduce explores a method to use locality as an additional resource. The idea is for the distributed file system to place data as close as possible to where they will transformed — it will try to store map and reduce data close to the mappers and reducers that will transform those data —, effectively diminishing the transport over the network.

Complexity

A priori, variables M and R, the number of partitions of the input space and of the intermediate space respectively, may be configured to take any value whatsoever. Yet, there exist certain practical limits to their values. For every running job the master will have to make $O(M + R)$ scheduling decisions — if no error forced the master to reschedule tasks —, as each partition of the input space will have to be submitted to a mapper and each intermediate partition will have to be transmitted to a reducer, coming to $O(M + R)$ as the expression of *temporal complexity*. Regarding *spatial complexity*, the master will have to maintain $O(M \cdot R)$ as piece of state in memory as the intermediate results of a map task may be propagated to every piece R of the reduce space.

Backup Tasks

A situation could arise in which a cluster node be executing map or reduce tasks much slower than it theoretically could. Such a circumstance may arise with a damaged drive which would cause read and write operations to slow down. Since jobs complete when all of its composing tasks had been finished, the faulted node (the *straggler*) would be curbing the global throughput. To alleviate this handicap, when only few tasks are left incomplete for a particular job, *Backup Task*s are created and submitted to additional workers, making a single task be executed concurrently as two task instances. By the time one copy of the task succeeded it would be labeled *completed*, duly reducing the impact of stragglers at the cost of wasting computational resources.

Combiner Function

Many times it happens that there exists a good number of repeated intermediate pairs. Taking wordcount as an example, it can be easily seen that every mapper will generate as many tuples *("a", "1")* as *a*'s there are in the input documents. A mechanism to lower the tuples that will have to be emitted to reducers is to allow for the definition of a *Combiner Function* to group outputs from the map function, in the same mapper node, before sending them out over the network, effectively cutting down traffic.

In fact, it is usual for both combiner and reduce functions to share the same implementation, even though the former writes its output to local disk while the latter writes directly to the distributed file system.

2.2.6 Other MapReduce Implementations

Since 2004 multiple frameworks that implement the ideas exposed in the paper [2] have been coming out. The next listing clearly shows the impact MapReduce has created.

Hadoop [8] One of the first implementations to cover the MapReduce processing model and framework of reference to other MapReduce codifications. It is by far the most widely deployed, tested, configured and profiled today.

GridGain [9] Commercial and centered around in-memory processing to speedup execution: lower data access latency at the expense of smaller I/O space.

Twister [10] Developed as a research project in the University of Indiana, tries to separate and abstract the common parts required to run MapReduce workflows in order to keep them longer in the cluster's distributed memory. With such an approach, the time taken to configure mappers and reducers in multiple executions is lowered by doing their setup

only once. *Twister* really shines in executing *iterative* MapRecude jobs — those jobs where maps and reduces do not finish in a single map-reduce sequence, but need instead a multitude of map-reduce cycles to complete.

GATK [11] Used for genetic research to sequence and evaluate DNA fragments from multiple species.

Qizmt [12] Written in C# and deployed for MySpace.

misco [13] Written 100% Python and based on previous work at Nokia, it is posed as a MapReduce implementation capable of running in mobile devices.

Peregrine [14] By optimizing how intermediate results are transformed and by passing every I/O operation through an asynchronous queue, its developers claim to have formidably accelerated task execution rate.

Mars [15] Implemented in *NVIDIA CUDA*, it revolves around extracting higher performance by moving the map and reduce operations into the graphics card. It is supposed to improve processing throughput by over an order of magnitude.

Hadoop is undoubtedly the most used MapReduce implementation nowadays. Its open-source nature and its flexibility, both for processing and storage, have been reporting back an increasing interest from the IT industry. This has brought out many pluggable extensions that enhance Hadoop's applicability and usage.

Chapter 3

Experimental Assessment of IaaS Clouds

In this chapter we will be reviewing the most used frameworks to drive IaaS Clouds. An initial selection will be made and it will be progressively shrunk following certain criteria like maturity, ease of use or documentation quality, until only one remained. A deep study will be carried out on the prevailing last.

3.1 Assessment Methodology

A thorough evaluation of the capabilities of the different frameworks is not possible unless an actual deployment is carried through. The virtual infrastructure that is generated when a cloud has finished installing, no matter how small the deployment be, is large and complex. Besides, *emulating* the hardware that will support the cloud is meager most of the times. *Nested Hardware Virtualization* — the capacity for a CPU to export its native virtualization capabilities to a guest running atop a host node, or the ability to use full hardware virtualization *inside* a virtual machine —, does not currently enjoy widespread adoption as it requires implementation efforts from both CPU designers and virtualization software developers. This means that it will not be possible for us to fully appraise the myriad IaaS Cloud solutions by creating a virtual cluster over which to deploy our clouds, and make performance measures.

To diagnose the superiority of one of them over the rest, a scaled down setup will be completed to evaluate the proficiency in maintaining the IaaS service running in spite of the reduced infrastructure. Quality and transparency in the documentation, as well as community support and engagement will also be borne in mind.

The testing environment follows the organization shown in figure 3.1.

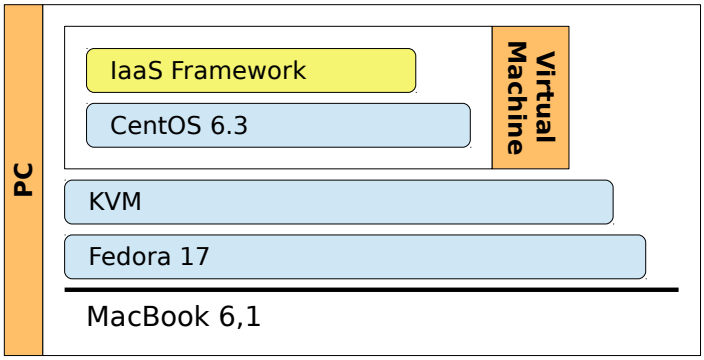

Figure 3.1: General testing environment

3.2 Evaluated Frameworks

The frameworks are:

- CloudStack
- Eucalyptus
- OpenNebula
- OpenStack

From an exclusively functional vantage point, the four of them clearly cover the requirements to run and manage the life cycle of an indeterminate number of custom VMs, through an API that would allow for the definition of a job control interface.

Eucalyptus and *OpenStack* take a more modular approach to the solution, unveiling smaller functional parts grouped in modules, while at the same time decoupling those modules from each other. With a module set in this fashion, installations become more flexible and tougher on par. However, to contain the operative effort, OpenStack ships with a series of scripts that help managing its deployments. Regarding their system requirements, they all support installations in modern Linux distributions with KVM or Xen as hypervisors. When dealing with a real deployment, which framework will be chosen will likely depend more on the existing platform than on particular limitations that any cloud may have.

As a side note, it is remarkable the lack of interoperability among them. All of them try to adhere to the AWS API in a different degree — some of them support it partially, others use *adaptors* to bridge their implementation to an AWS-compatible API. OpenNebula, OpenStack and Eucalyptus have demonstrated to be carrying on coding efforts to fully support the OGF's standard: OCCI.

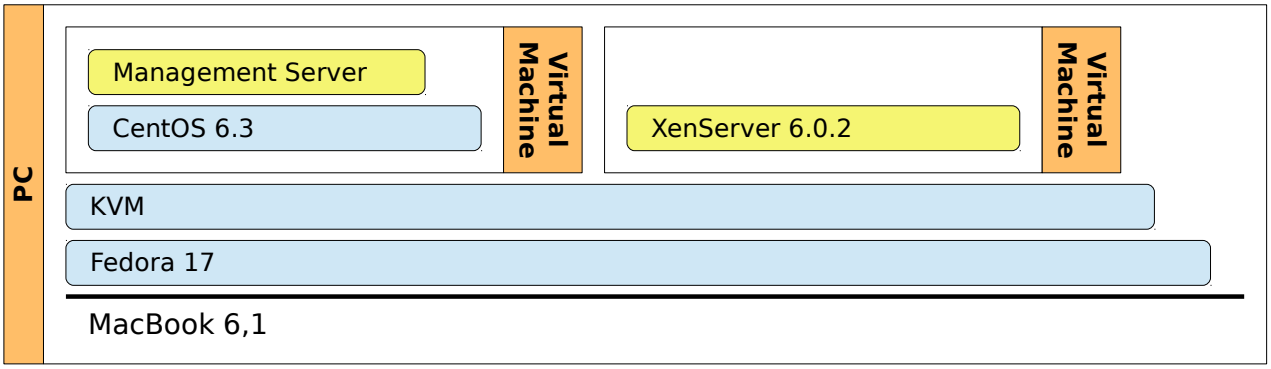

Figure 3.2: CloudStack 3.0.2 with XenServer hypervisor

Eucalyptus, in spite of being the first to fully cover the AWS API, which is merely anecdotical nowadays, has two obstacles that hinder its evaluation. First, it is not fully open-sourced: `VMware Broker` which brings the opportunity to use virtualized infrastructure based on VMware technology, is only available to paid subscribers. And second, it is impossible to setup Eucalyptus within a VM to test start-up time or installation complexity, for example, as it explains its installation guide [16]. Both limitations make Eucalyptus back out from the evaluation list. The rest have been compared after their installation and configuration.

3.2.1 CloudStack

CloudStack installation guide [17] describes the series of steps that a systems engineer should follow in order to complete a minimum CloudStack deployment. It clearly determines that Cloud Nodes' CPUs have to support virtualization extensions for CloudStack to start Xen or KVM-based VMs, which happens to be a similar limitation to Eucalyptus'. However, the fact that CloudStack would become part of *The Apache Software Foundation* from version 4 onward [18], and the reality of a Citrix technical article opening the door to CloudStack deployments over Cloud Nodes lacking virtualization extensions [19], made us arrange the layout shown in figure 3.2 to test-drive the CloudStack cloud implementation.

Following the advanced and quick installation guides — [17] and [20] — the process may be summarized in the steps below:

- Two VMs were created to contain the CloudStack *Management Server* (*MS*) and the XenServer hypervisor: 1 GB of RAM for the MS, 3 GB of RAM for Xen, 20 GB HDD, *ACPI* and *APIC* for both.
- For the MS:
  - *CentOS 6.3* was downloaded, installed and `yum-updated`.

  - The VM was named *cloudstack*.
  - Likewise, a user named *cloudstack* was registered and added to the *sudoers* list.
  - The quick installation guide was followed to conclude the process.

- For Xen:
  - XenServer 6.0.2 was downloaded from Citrix web site.
  - The notes section in the quick installation guide and in the XenServer configuration manual [21] were followed to perform the configuration.

- Additionally:
  - Before specifying the execution environment a global flag had to be set to permit nodes with no virtualization extensions [22] be part of the Cloud Nodes set.
  - Once the configured infrastructure was online:
    * The CentOS 6.3 image was uploaded to the MS.
    * A `SimpleHTTPServer` — a Python micro HTTP server — was started on port *443* in the MS.
    * A rule was added in `iptables` to let traffic through on port *443* in the MS.
    * The VM image was loaded to the cloud from CloudStack's web interface.

3.2.2 OpenNebula

If balanced against the installation procedure just described, the effort for setting up OpenNebula 3.8 is lighter. The process that has been followed to configure the OpenNebula deployment contained in figure 3.3 stems directly from [23], the official installation guide. The subsequent steps serve as summary to the process.

- A supporting VM was created to fully contain OpenNebula: 1 GB of RAM, 8 GB for the HDD, ACPI and APIC.
- CentOS 6.3 was downloaded, installed and yum-updated.
- The VM was named *opennebula*.
- A user named *opennebula* was also registered and added to the sudoers list.
- `SELinux` was stopped.

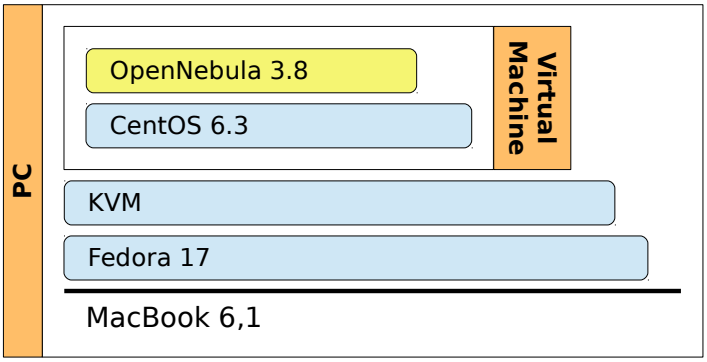

Figure 3.3: OpenNebula 3.8 with KVM

- The configuration guide [23] was followed to complete the process. Afterwards, the next series of actions were carried out to be able to test the framework:
  - By default, OpenNebula's web interface module (`sunstone`) attaches to *lo*. In order to interact with sunstone from outside of the containing VM, it was necessary to change the configuration file (`/etc/one`) so that sunstone attached to the external network interface *eth0*. The very same happened with `occi`, the REST service that exposes the cloud API.
  - Furthermore, iptables was modified for letting traffic through on port *9869*; where sunstone listens by default.

3.2.3 OpenStack

OpenStack case is striking for many reasons. It represents the convergence of two different needs: the computationally-driven in NASA and the storage-bound in Rackspace. Complementarily, both Red Hat and Canonical had shown their interest in the platform by collaborating with their scripts, deploying utilities and cloud-optimized images.

Just as with OpenNebula, the complete execution environment was installed into a single VM. In this case, Fedora was chosen over CentOS or Ubuntu for the existence of a community-written installation guide [24] and scripting utilities to ease the process.

Both Essex and Folsom versions were tested. The reasoning behind this is that in spite of being Essex the officially supported version in Fedora 17, the quick installation guide suggested using the latest version available — Folsom by December 2012 — enabling a testing repository to that end. The degree in maturity observed moving from Folsom to Essex was startling: not only the web interface had been revamped giving it a more thorough look that also reflected much better the state of the underlying infrastructure, under the hood, the core module had

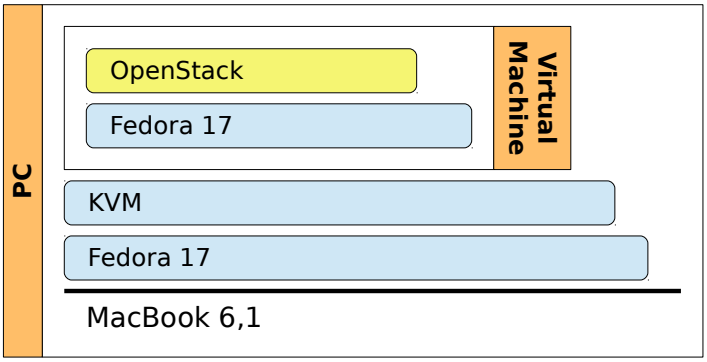

Figure 3.4: Virtual OpenStack deployment

also been split in smaller functional pieces that could be easily distributed across the cluster. In Essex, when dealing with creating instances in the cloud, if there were any problem while the networking interfaces were being brought up, making it impossible for them to obtain IP addresses, the web interface would hang in a state that would not allow destroying the instance, requiring the invocation of console commands to fix the issue. This problem, for instance, is solved in Folsom.

An execution environment for both versions was spawned by following the next list of actions (see figure 3.4):

- A single VM was created to hold OpenStack: 1 GB of RAM, 10 GB for the HDD, APIC and ACPI.
- Fedora 17 was downloaded, installed and yum-updated as usual.
- The VM was named *openstack*.
- The *openstack* user was registered as well as added to the sudoers list.
- `acpid` package was installed.
- SELinux was stopped.
- A full clone of the VM — which includes the virtual drive — was made to test both versions.
- The quick installation guide as well as the official one, omitting Swift, were followed to complete the process.

3.3 Veredict

What follows is the listing with the findings in comparing the frameworks according to the methodology described at the beginning of this section.

Installation: Without a doubt OpenNebula stands out in this regard. The installation guide is the lightest and shortest to follow with a difference. It carries, however, the inherent problem of hiding what is going on when it is being configured for the first time, potentially hardening the resolution of issues that might appear in the future.

Configuration and Management: Growing the supporting cluster requires, on every cloud tested, that a compatible hypervisor be installed on any node added. CloudStack and OpenNebula offer a more transparent management interface to better control the physical infrastructure. OpenStack displays the most limited web interface.

Hypervisor: Regarding hypervisor support, OpenStack clearly surpasses both CloudStack and OpenNebula. Yet, they all support the most widely used hypervisors — KVM, Xen, Xen variants and VMware and variants — in production deployments.

Storage: The three of them support a broad assortment of data back-end controllers. But, in this case, it is important to highlight the effort OpenStack is involved in to introduce Amazon S3 compatibility in its deployments. Swift is the OpenStack component granting fault tolerance and high availability storage, mimicking Amazon S3, relying on data replication and balancing among other techniques. CloudStack advanced installation guide [20] describes a first approach toward configuring Swift as secondary storage for the cloud. This fact speaks volumes about the maturity and importance of Swift.

Documentation: None of the three can boast about exhaustive official installation guides. Every framework has had its own exposure to different Linux distributions, so the coverage they offer varies to the point of mistaking module names; e.g., both CloudStack manuals are more easily followed using CentOS as base operating system and XenServer as hypervisor. OpenStack provides installation manuals supporting both Red Hat and Debian derivatives, but for Fedora the name of the services in the documentation does not correspond to the real ones; no such thing happened for Ubuntu. Nonetheless, inaccuracies are trivial to cope with and the manuals are deep enough to be used to make successfully deployments in production environments.

Community: Even though it may seem unimportant, the community is vital for developing and supporting the frameworks. They are, at the very least, partially open-sourced, so a lively community translates into higher usage rates, more rigorous documentation, more

bugs squashed, etc. While it is hard to assess the magnitude of an online community from the outside, it is interesting to highlight the nourishing that OpenStack is continuously receiving from Red Hat and Canonical: there is no technical keynote or conference in which OpenStack is not appointed.

3.3.1 OpenStack Folsom

The IaaS Cloud that has been chosen is OpenStack Folsom. The lengthy installation guides, the community support, the backing by two large software companies, the real deployments in production (from HP, Dell, Intel, Rackspace, etc.), the modular configuration, the completeness of the implementation (OCCI APIs, S3, EC2, Swift, etc.) and the *official* support to deploying clouds over virtual clusters for testing have unbalanced the comparison in its favor.

Chapter 4

OpenStack Folsom

The current section intends to detail the IaaS Cloud implementation that has been chosen: OpenStack. Initially, a global vision will be given to the reader, to progressively focus on its constituent modules' responsibilities and how they collaborate to maintain the service running.

4.1 Global Architecture

Figure 4.1 shows the three basic operational components of OpenStack Folsom:

Functional Core: OpenStack Compute, OpenStack Quantum and OpenStack Storage (Cinder and Swift).

Web Management Interface: OpenStack Horizon.

Shared Services: OpenStack Glance, OpenStack Keystone and other related services like a DBMS for persisting meta-data or a messaging queue.

The different components have been devised to run in a shared-nothing fashion. This provides the cloud admin the flexibility required to distribute the modules over the cluster as pleased. An example of a particular OpenStack deployment is shown in figure 4.2. Missing from the diagram, for clarity, are the connections between the different modules and the asynchronous queue that mediates their communication. Qpid and RabbitMQ are the two queue implementations that are officially documented, being the former the one used in our test deployment.

4.2 Horizon

Horizon represents a window to configure and monitor the cloud. As discussed in the previous section, Horizon does not currently — as of Folsom version — present a global view of the

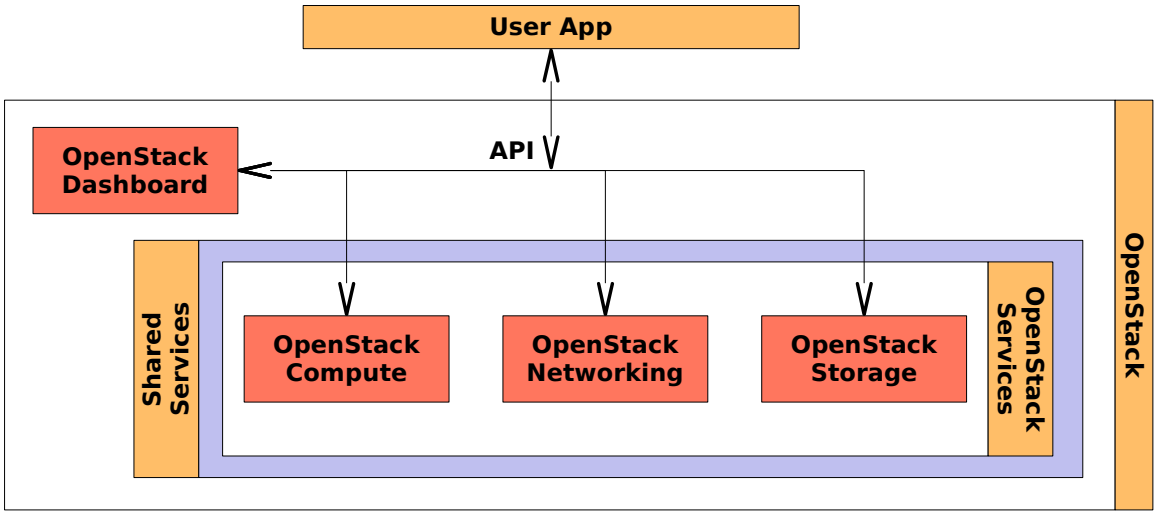

Figure 4.1: OpenStack Arquitecture

physical infrastructure, leaving the user in the dark in this respect. Horizon is written in Python on top of `Django`, a web framework. Django itself relays on a web server like `httpd` to expose static files, uses a caching mechanism (`memcached`) to speedup load times and a terminal embedding system (`noVNC`) to view the output of the virtual graphics card directly on Horizon.

To manage and create instances in the cloud, OpenStack gives the cloud admin the ability to register authorization roles that will let the users consume those services whose roles gave access to. While the cloud administrator is allowed to register custom roles, two roles that ship with the distribution are the *Cloud Admin* and the *Cloud Member*.

A user granted the Admin role will be able to manage:

Tenants: Create, delete, member users, alter quotas, etc.

Users: Create, modify or delete.

OS Images: List, remove or modify meta-data.

Instances: Reset, shutdown, suspend, print log on screen, etc.

Volumes: Create, list, attach to an instance, etc.

Networks: Create, modify or delete.

A user granted the Member role will be able to:

Status: Quota, resources, etc.

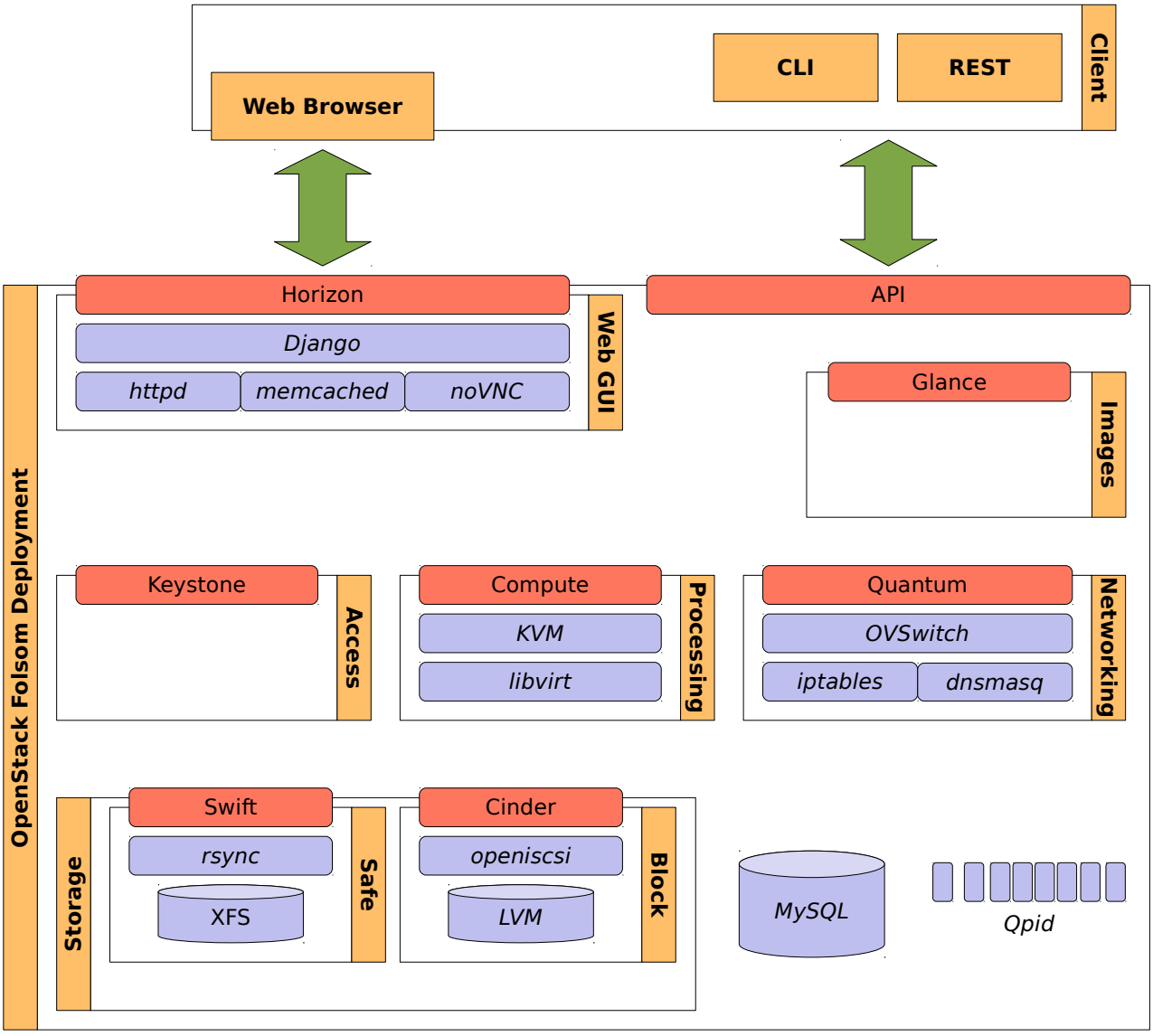

Figure 4.2: Example of an OpenStack Folsom deployment

Instances: Create, shutdown, reset, suspend, print log, create image from a running instance (snapshot), etc.

Volumes: List, create, modify, attach to an instance, create a volume snapshot, etc.

Images: Create, list, delete, modify, etc.

Networking: Manage public IPs (floating IPs).

Security groups: Create, delete or modify security rules.

Keypairs: Create, modify or delete.

4.3 Keystone

Keystone is the central security check-point and user-related data repository, storing the information needed to access the cloud's installed services. It verifies, before each request, user credentials and authorizations to OpenStack services. Keystone divides this functionality in two parts: user control and service catalog.

To organize users, Keystone assigns them to tenants or projects. Users, as discussed above, are granted the membership to a tenant and a service quota they will have to adhere to; they are also restricted to the tenant's quota.

To organize the catalog at hand, Keystone defines two other concepts within the service catalog: *Services* and *endpoints*. A service in the catalog is a mere abstract description of an exploitable cloud feature by the user. The particular implementation of the service is managed by the set of endpoints associated with it. Said collection contains every piece of information that is required for users to consume the services. Figure 4.3 shows a Sequence Diagram portraying the interchanged messages between the different entities taking part in the consumption of the service *Create new instance*.

Stemming from the fact that Horizon exposes only a part of OpenStack functionality, to help dealing with security Keystone installs a CLI tool to interact with the REST service dealing with administrative operations. Issuing certain commands to Keystone through a terminal application requires the knowledge of the admin token, which has to be conveniently secured, or the login credentials of a user with the Admin role. Lastly, it should be noted that Keystone uses a data base to store user access credentials and the service catalog meta-data.

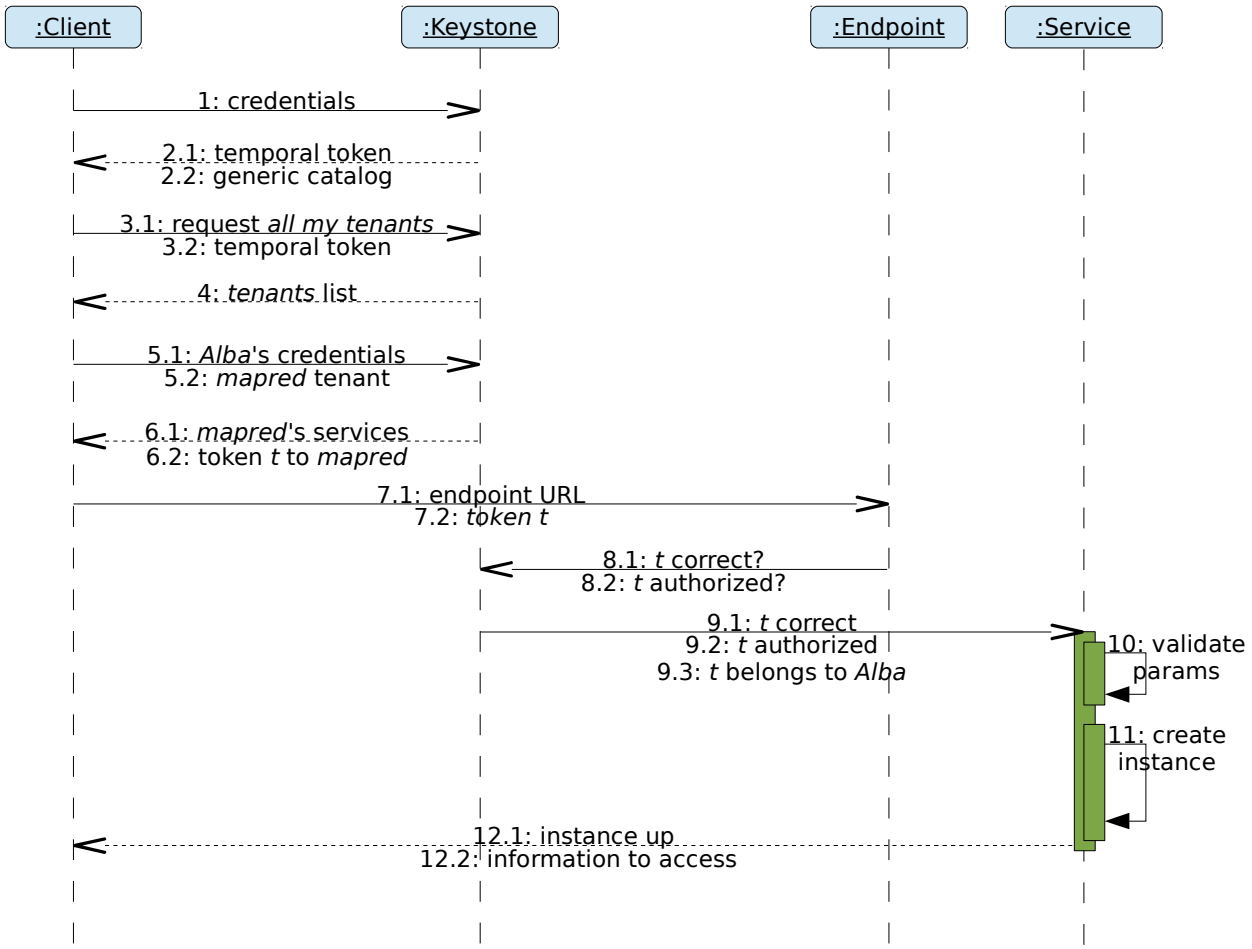

Figure 4.3: Sequence Diagram — create instance

4.4 Quantum

Starting with Folsom, Quantum is the module to manage virtual networking. It was introduced to separate the networking part from the computing part, hitherto held together in the `Compute` module. Certainly, the fact that it had been refactored out demonstrates OpenStack's evolving model toward a more coherent and less coupled functional allocation between modules.

To bring virtual networking into existence, Quantum banks on external plug-ins. Two of those plug-ins whose usage and configuration is covered in the official Quantum administrator manual [26] are `OpenVSwitch` and `LinuxBridge`. Additionally, Quantum relies on iptables to configure routing rules and firewall, `dnsmasq` for *DNS*, *DHCP* and *NAT*.

Figure 4.4 pictures the example of a virtual network configured with Quantum. On it, *30.0.0.X* represents public IPs and *10.0.X.Y* private IPs. This virtual network assigns only a single virtual router to each tenant but more networking entities could be added with ease. Private IP overlapping over different networks is possible as expected (*10.0.0.2*). The routers' public IPs — they could be attached more than one external interface — must be taken from the external network (*30.0.0.0*).

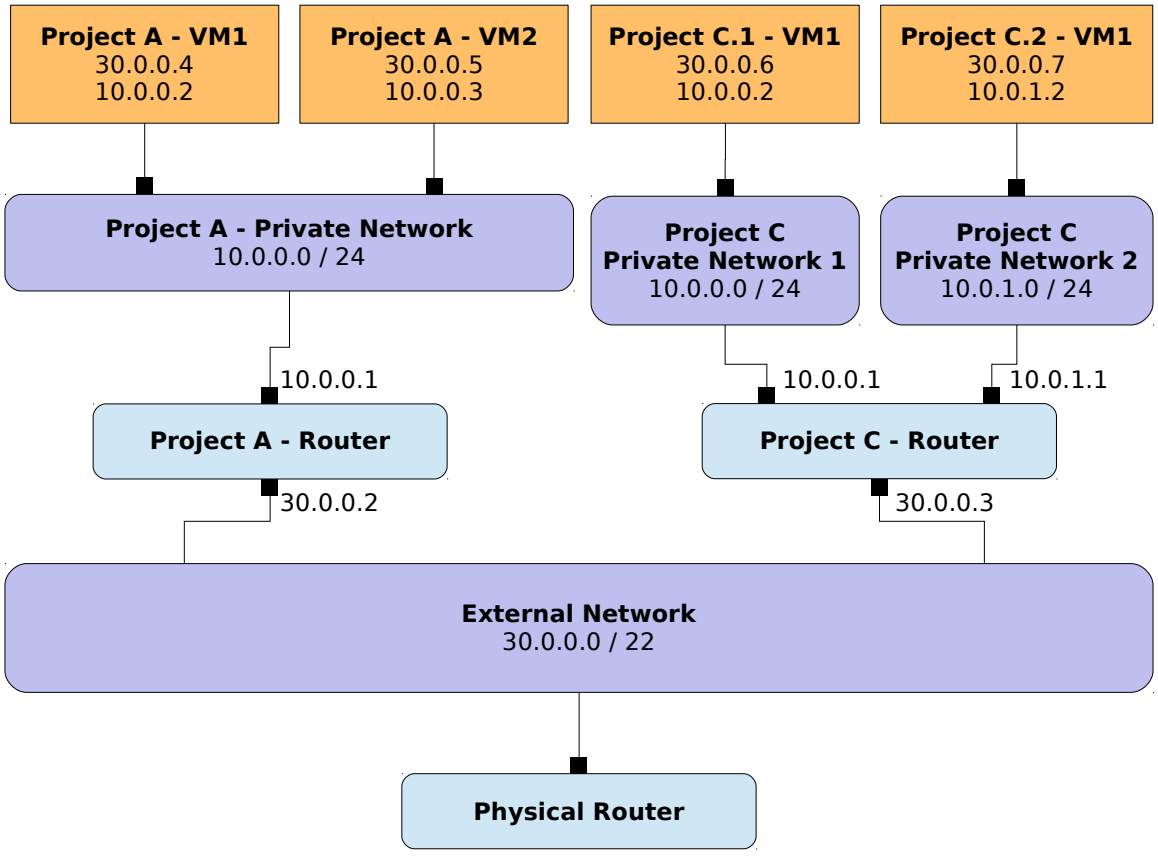

Figure 4.4: Virtual network deployment with Quantum

4.5 Compute

Compute is the central module. Its duty entails orchestrating the global inner workings of the cloud, delegating each particular function to the module on charge. In the end, Compute will let a logged user start virtual instances, which will draw their VCPU, VRAM and VHDD from the physical cluster. Yet required to function, Compute does not contain a virtualization package. The approach is to delegate infrastructure provision to a hypervisor typically found, but not restricted to, in the same node. To expose this on-demand computational service, Compute implements a REST API so that users can control their instances' life cycle directly from a REST client (like the CLI tools that accompany Compute which interact with the service through its REST interface).

To effectively create an instance, Compute will collaborate with other modules within the cloud to orchestrate the execution in the *most suitable* cluster node hypervisor — most suitable according to a cloud-defined rule set. Some of the supporting services are described below.

Keystone: Collates credentials y authorizes requests.

Glance: Selects the OS image that will be used to start the VM.

Quantum: Grants private and public IPs as well as manages instance network traffic.

Cinder: Manages block storage and on-line volume attachments.

Qpid: Handles message interchange between Keystone, Quantum, Glance and/or Cinder.

As it has been discussed all along, if there is some trait that aligns different IaaS Cloud implementations is their flexibility. Users' computational needs are as diverse as they are changing and therefore they expect to be given the chance to define virtual infrastructure adapting to those needs. In OpenStack, each possible particular configuration instance will take its VCPU, RAM and VHDD from a cluster host, and the users will be allowed to shape those variables to their requirements with ease.

4.6 Glance

Glance is OpenStack's VM image storage service. Glance may be configured to drive images stored in a myriad of backends, ranging from Swift to an HTTP-addressable location. As it happens with every other OpenStack module, Glance relies on Keystone to grant access to the images, and coordinates its operation with Compute to put them in execution on-demand.

Glance supports a good number of image and container types — this fact being merely informative as it is the hypervisor which would have to support the particular combination of image type and container type —, and they are stored as meta-data to the image in Glance.

4.7 Storage

OpenStack provides three main options regarding storage types:

Ephemeral: The size of the drive hosting the root file system is set following the particular *flavor* parameters when the VM starts. The files contained in this file system are those present in the image file stored in Glance. Any alteration to this file system will only persist with the execution of the VM. Any change to the image file system is written temporarily to be discarded as soon as the VM shuts down.

Block: By making use of storage volumes managed by Cinder with *LVM* (*Logical Volume Manager*) OpenStack provides the ability to attach indeterminately-sized logical volumes to instances on-demand. This kind of storage guarantees that information is preserved between VM executions. However, this method carries an important handicap, that of being unable to attach a single volume to two different instances at same the time. High availability or data safety on failure are not supported, as data is stored in a single place. A backup or RAID policies may be established to get over these limitations but they are discouraged as OpenStack has it own module to deal with them: Swift.

Safe: Swift manages a safe distributed storage banking on controlled replication allowing for high availability deployments that overcome the hard drive's inherent fragility.

4.7.1 Cinder

Cinder is the OpenStack module that takes care of virtual block storage devices — functionally similar to Amazon's *EBS* (*Elastic Block Storage*). Cinder uses an *iSCSI* implementation (`open-iscsi`) and LVM to manage operations on the volumes. Creation, attachment and detachment, and logical volume removal is directly controlled through Horizon.

Those persistent virtual blocks are administered as logical volumes pertaining to a volume group controlled by Cinder. Cinder, though, shall not be used to create a shared medium for instances, as *NFS* (*Network File System*) or *SAN* (*Storage Area Network*) solutions do; for a single volume cannot be attached to different instances at the same time.

4.7.2 Swift

Just as happened with Cinder, Swift cannot be framed into traditional shared networking applications nor be compared to Cinder: Swift covers a different functional demand. Swift is defined as "*a scalable object storage system where logged users control their store buckets uploading, downloading or deleting files to their will*" [27]. Swift may be conceived as a functional clone to Amazon's S3 or Eucaliptus' `Walrus`, implementing a REST API partially compatible with Amazon's. Central to Swift's implementation is replication.

Replication

Scalability, fault tolerance, high availability, safe storage and load balancing are some distinguishing features of Swift's. As discussed, high availability and fault tolerance are implemented with replication. Replication is a mechanism by which a distributed system keeps copies of file system blocks at different locations to guarantee better performance and limit failure impact.

Within Swift, replication processes on every *Object Server* — any node in the cluster configured to support Swift storage — periodically compare their local blocks with remote replicas to collate their update state. Comparing replicas' states is as costly a process as it is often, thus *hash lists* and *watermark*s are used to improve comparison time. Replication is transparent to the user and Rsync or HTTP transport replica payload across the cluster. When a new Swift node is added to the cluster, replica distribution becomes unbalanced and Swift will trigger an automatic rebalancing procedure. When it be synchronized with the cluster, the new node will be able to respond to data requests.

Updaters y Auditors

Other supporting services that complete the functional circle defined for Swift are *Updater*s and *Auditor*s. The former act when a replica synchronization error is raised or when Object Servers load is so high as to make a data request not be responded. It happens then that the execution of this data request is delayed and queued, being serviced by the Updater process at a later time. Auditors continually scan the file system looking for integrity failures in objects or buckets. If an inconsistency were to be found, the incoherent entity would be quarantined and replicas would be made anew.

Chapter 5

Hadoop

This chapter will try to expose the defining fundamentals of Hadoop architecture. General concepts will be introduced first, to give way to deeper ones that will help explore Hadoop's implementation of the MapReduce paradigm in two layers: the processing model and the storage subsystem.

5.1 The Beginnings

Hadoop roots its origins in *Apache Nutch*: Mike Cafarella and Doug Cutting's implementation of an open source web index and search engine. The Nutch project began in 2002. In spite of the Internet being notoriously smaller at the time, Nutch's underlying technology was unable to make it scale to manage the billion pages that comprised that *reduced* Internet. But in 2003 Google publishes a research paper introducing *GFS* (*Google File System*) [28], a file system to be used across their clusters of commodity machines that greatly simplified storing and recovering data from them. Nutch will come to inherit a large part of the concepts detailed there, translating in their own distributed file system implementation (*NDFS*).

Also in 2004, there appeared another Google publication [2] that presented the MapReduce execution paradigm. This paper would bring about successive efforts to port Nutch's algorithms to the recently introduced model. In mid 2005, most of Nutch workflows could be run as MapReduce workflows over NDFS.

Both NDFS and the Nutch MapReduce implementation were generic enough to be used beyond web page indexing without refactoring. In 2006, an unrelated project was started to extend Nutch's potentially reusable parts and widen its applicability context. This project was called Hadoop. In 2008 *Yahoo!* announced that their web index was being continually refreshed by a 10,000-node Hadoop cluster. This same year, Hadoop is brought out to the world becoming an Apache-backed project.

Nowadays, Hadoop is without a doubt the MapReduce implementation most widely used in a broad range of applications.

5.2 General Hadoop Architecture

Hadoop architecture separates four modules:

Hadoop Common: A module containing the parts used across the implementation. It is mainly comprised of scripts and configuration tools.

Hadoop MapReduce: the module implementing the MapReduce processing model.

Hadoop YARN: A general-purpose framework abstracted from the *classical* Hadoop MapReduce. It is employed to manage computational resources and schedule executions in distributed environments.

Hadoop DFS: The distributed file system sustaining I/O operations and storage for Hadoop clusters by default.

Hadoop architecture corresponds to the *Master-Worker* archetype: in the clusters of machines there will be two operating roles: a unique Master and various Workers. These roles are set during daemon start-up and may not change unless they be rebooted. If necessary, e.g. for maintenance, the cluster admin may freely re-set the roles to new cluster nodes, only requiring job resubmissions if the Master role were reassigned *and* pending jobs were present.

Hadoop YARN is a subsystem resulting from the isolation of scheduling and processing, coupled together in the first MapReduce library, separating task distribution and planning into YARN. In this way, it may be possible for Hadoop deployments to process implementation-agnostic working sets.

Figures 5.1 and 5.2 exhibit a high level vision of Hadoop architecture. Figure 5.1 shows an hypothetical deployment with HDFS. Figure 5.2 shows a particular Hadoop installation with another distributed file system as storage layer.

As exposed in the figures, Hadoop runs atop a *Java Virtual Machine* (*JVM*) and inter-node communication is securely conveyed through *SSH* tunnels. Besides, every Hadoop module includes a web server (*Jetty*) to ease collecting and reporting status information with standard tools like web browsers or text processing CLI applications.

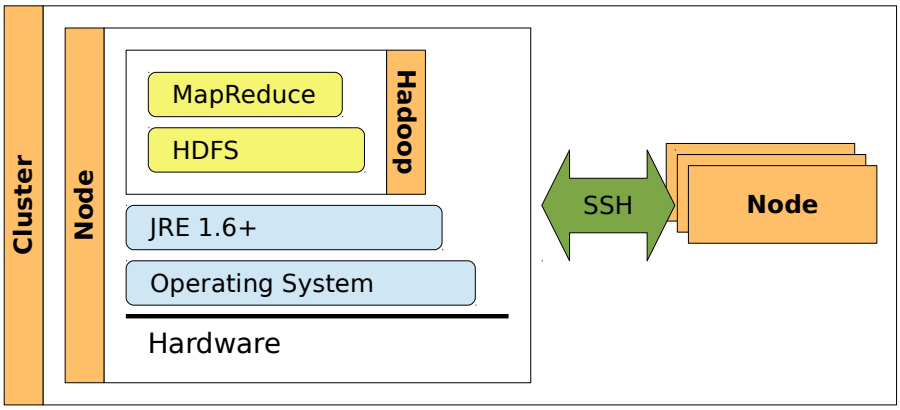

Figure 5.1: Hadoop over HDFS

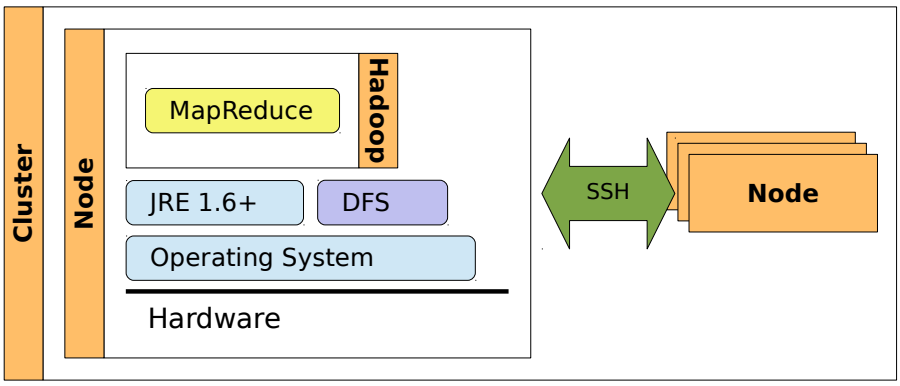

Figure 5.2: Hadoop over another DFS

5.3 Hadoop Distributed File System

HDFS has been designed to act as archival repository for huge masses of data whose main access pattern be *write-once read-many*. While it is no requisite for a data query to follow this access pattern, HDFS performance shines with read queries in batches. The underlying infrastructure, again, is composed by commodity machines that HDFS will manage to transform into a reliable, scalable, fault tolerant, self-balancing and network traffic-reducing data store. Yet, as individual nodes may be regular computers, in HDFS converge some operating limitations:

- High *data access time*. HDFS prioritizes large reads in batches and thus, reading small files is discouraged. However, HDFS delivers *very* high throughput by leveraging parallel reads on the cluster.
- High *data write time* on many small files. Writing a file changes a block. Every file that be smaller than the HDFS block size must be persisted in a single block. That changed block must be sent out across the network to keep the file system coherently updated. Thus, appending to n files may at least require updating n blocks that would need be synchronized over the network.
- Multiple writers to the same file or *not-append* file operations are not supported. HDFS is not a *POSIX*-compilant file system as it implements only the set of operations required to maximize data throughput in distributed environments.

To organize storage HDFS takes from traditional file systems the concept of *block*. In this case, an HDFS block is an abstraction atop the particular local file system with a double purpose:

Lower DFS complexity: Writing a block comprises storing the block data and meta-data, and handling information on how to locate the block on disk. Using the block as the minimum organizational structure simplifies the algorithms to manipulate de file system.

Increment flexibility: files are free to grow over the size of an HDFS block with no penalty on access time or throughput.

To lay blocks in position, HDFS makes use of two processes: the *DataNode* and the *NameNode*. Besides, as support, from Hadoop 0.21.0 onward it is permitted the optional deploying of a *Backup Node* and a *Checkpoint Node* in the same cluster.

5.3.1 Node Roles

Figure 5.3 shows a layered down HDFS deployment. In dotted line appear both the Backup Node and the Checkpoint Node.

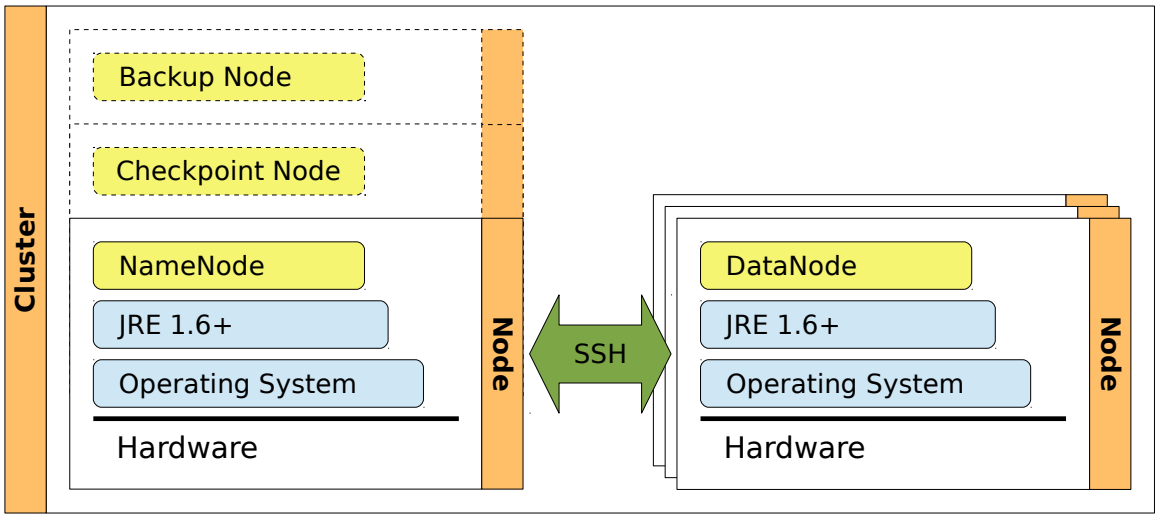

Figure 5.3: Typical HDFS deployment

DataNode

DataNodes are those processes within nodes that handle the storage of HDFS blocks in their local drives. Every time a write to a file in a block succeeds, the DataNode in charge of the operation signals its supervising NameNode so that it can keep track the modifications to the DFS as they happen.

NameNode

The NameNode is the process appointed to deal with the name space of the cluster. It has to handle the file system tree and meta-data that make possible recovering data from the DFS. It is such an important process that if it went down, every piece of data in HDFS would get lost, rendering impossible to match files with their container blocks. Therefore, the NameNode is seldom deployed with no Checkpoint Nodes or Backup Nodes when no other backup policity is implemented.

The information about the file system, the meta-data, is persisted to the NameNode both in-memory and disk. In this latter form, the meta-data is managed in two files: one contains the name space of the file system as an image (`fsimage`), the other progressively appends the changes to the fsimage (*edits*) as a log. When the NameNode starts, it compiles an fsimage afresh by merging the existing fsimage with the edits file. As soon as changes to the HDFS are reported from the DataNodes, the NameNode will update the edits log but without touching the fsimage. In a typical secured deployment, the edits file is kept in-memory and in the local file system in the NameNode computer, and remotely via NFS.

Checkpoint Node

The Checkpoint Node is the means by which failure in the NameNode poses a smaller threat to the integrity of the data within HDFS. The idea behind the Checkpoint Node is to maintain a separate copy of the edits file so that it could periodically merge the fsimage with the edits, generating an updated fsimage to be uploaded to the NameNode. It should be noted that the Checkpoint Node does not listen to the network to record the modifications to the DFS in the edits file, it fetches the copy held by the NameNode itself. When the merging process be finished, the NameNode will be submitted the newly created fsimage, clearing out the old copy and reseting the edits file.

Backup Node

The Backup Node provides the same functionality as the Checkpoint Node by periodically creating restoration points of the name space but through a different approach. In this case, the Backup Node will download the fsimage file from the *backed up* NameNode when booted, just like the Checkpoint Node, but it will receive notifications from the DataNodes on each modification to the HDFS, and will manage itself the merging between the fsimage and the edits file, effectively creating a new fsimage; the same behavior as the NameNode.

Compared to the Checkpoint Node this process draws less bandwidth due to the fact that it does not require to download the fsimage nor the edits file from the NameNode in order to stay synchronized.

The Hadoop version used in the VM of the solution, as will be described in section 6, allows only a Backup Node per NameNode or multiple Checkpoint Nodes, not both. It should be noted that including a Backup Node in a cluster gives the possibility to run the NameNode with no persistent storage, i.e. with no allocated space in the local drive to write the fsimage and the edits file, making the Backup Node the sole responsible for the duty.

5.3.2 Network Topology

One of the fundamental parts to a file system in a distributed environment is the capacity to provide a transparent mechanism by which information is persisted securely while, at the same time, a certain amount of performance is maintained. HDFS makes use of an already discussed technique, *replication*, that provides high performance, scalability and fault tolerance. Besides, HDFS has been designed to reduce the network congestion that stems from regular operation, giving special emphasis to the way in which data is to be distributed in the cluster, controlling replica count or distance between replicas, among other variables, to concrete the block allocation policy.

Node Distance

To support the features that have been mentioned, it is fundamental for the NameNode to possess some information on the participating DataNodes' physical location. With that information, the NameNode would be able to balance the *physical distance* of the replicas, bearing in mind that, as a general rule, fault tolerance increases with replica distance but the bandwidth available to transfer replicas narrows. Thus, the NameNode will have to distribute the replicas across the cluster with a procedure that kept the data in the cluster safe without clogging the network.

To effectively calculate the physical distance between two replicas, the NameNode will use an approximate measure of the physical distance of the nodes storing the replicas. Therefore, the problem lies in finding a formula to calculate the distance between any two nodes in the cluster.

IP-based networks arrange participating nodes in an abstract tree structure. In properly configured networks, the closer two nodes are physically the smaller the distance between their respective routing gateways is. This idea, along with the fact that IP networks are tree-shaped, may be used recursively until the *same* routing gateway was reached up from the initial nodes. So, the actual distance between two replicas is *the number of steps required to find a common routing gateway, starting from the two nodes that stored the replicas.* The example below will clarify the procedure.

Only using the distance among replicas to distribute them lacks an important feature that would render the cluster failure-prone. In order to keep HDFS fault tolerant, the NameNode has to deal with halting nodes, clogged networks, etc. If a stalled node were the gateway to a set of nodes, e.g the nodes in the same rack, then the access to any replica within that cluster would be impossible, and worse, if every replica of an individual block were stored in this rack, the block would be inaccessible for the duration of the outage in the gateway. Therefore, a complementary mechanism should be devised to solve this problems.

HDFS overcomes this limitations by mapping every IP address to a tuple with as many components as levels there were in the network — typically *three-tuples (data center, rack, node)*. Figure 5.4 shows the most common values of the replica distance. With the help of the figure, the procedure to get to $d = 4$ will be succinctly explained.

The upper left node holds the original block and the one pointed by the $d = 4$-labeled edge a replica. Both blocks are stored in nodes within the same data center but in different racks, thus, their nearest common gateway would be the one managing the routing between those two racks. Besides, every rack has an internal gateway that must be traversed in order for the internal traffic to exit the rack. So, each node would have to take *2 steps* to get to their common gateway — from node to *in-rack* gateway to *off-rack* gateway — coming to *4 steps*.

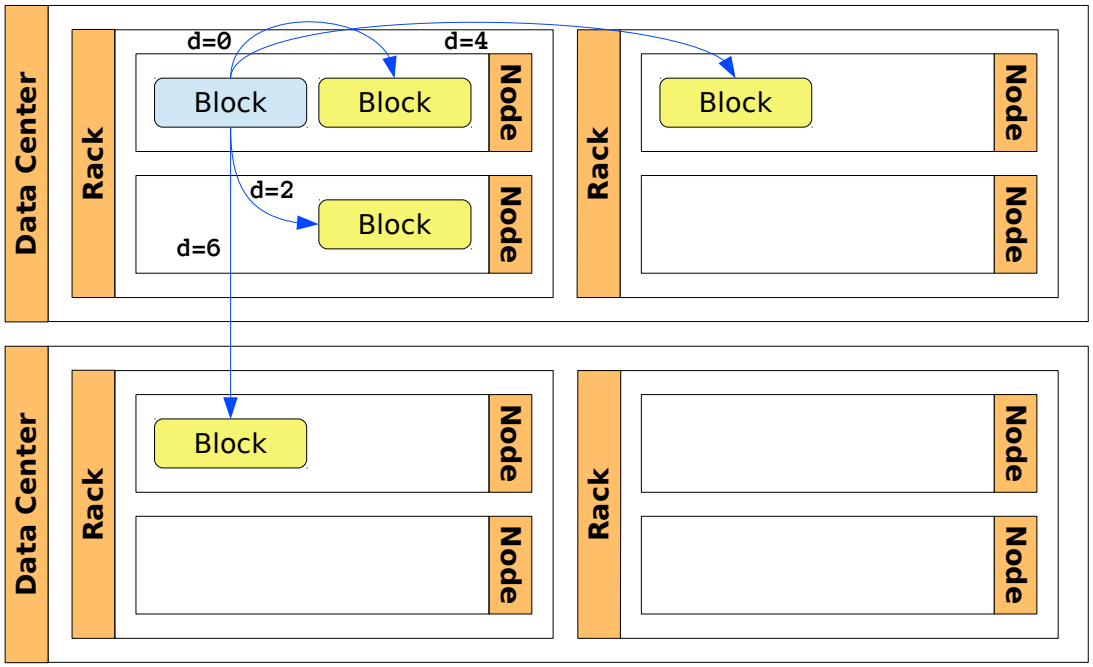

Figure 5.4: Replica distance

Replication

Replication is a NameNode-controlled technique that relies on knowing the network topology to structure hierarchically the nodes in the cluster. Together with replica distance the NameNode is capable of enforcing a policy that balances data safety and throughput. What follows is the description of the method to distribute replicas over the cluster.

When a new block is written to HDFS — or when an existing one is appended new data — a predefined number of replicas have to be made and distributed. By default, the NameNode will maintain updated three replicas of each block. The first block is placed in a node at random prioritizing those not too full nor too busy. Then, the first replica is stored in the same node as the original block. The second replica is sent off-rack to a node at random, and finally, the third replica is stored in the same rack as the second replica but in a different node. If the *replica factor* — the total number of replicas every block will have — were made higher, the successive replicas would be placed, again, in the same rack as the second one but in different nodes whenever possible.

This method provides the desirable equilibrium between fault tolerance — by assuring two copies of a block exist in different racks —, consumed bandwidth — writing a new block would only need replicas to go through a single gateway as subsequent replicas are made within the rack —, read performance — making it possible for every read request to be fulfilled from nodes in two racks — and balanced data distribution by executing the process just described. Figure

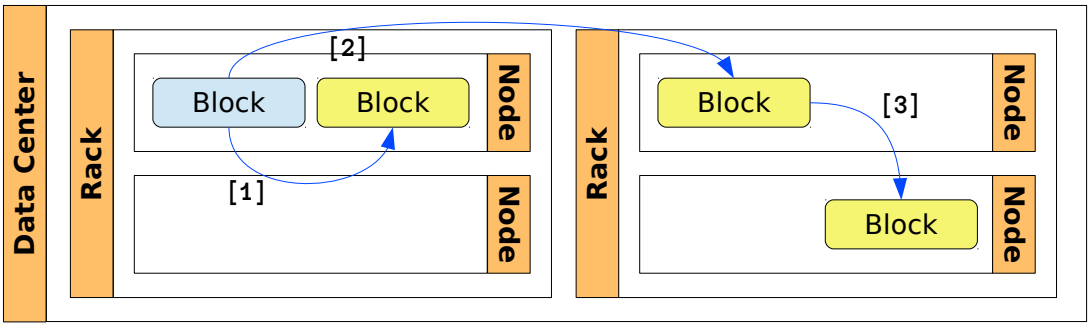

Figure 5.5: Replication factor 3

5.5 exemplifies a factor 3 replication. The number in brackets indicates the order in which the replicas were created. The original block is in the upper left corner of the figure.

5.4 Hadoop MapReduce

In a typical Hadoop deployment, the execution layer is lied over HDFS. This execution module is basically an implementation of the ideas exposed in the Google paper on MapReduce [2]. The core ideas that have been discussed for HDFS also hold true in this section, but with a different approach. Hadoop MapReduce architecture is also based on the Master-Worker model and so there appear two roles: a Master and a Worker.

The Master role is functionally covered by a process called the *JobTracker*, whose main responsibility is task scheduling to workers. Complementarily, the Worker role is implemented in the *TaskTracker* which will actually execute the individual tasks reporting their progress back to the JobTracker.

5.4.1 Node Roles

Figure 5.6 shows the JobTracker and the TaskTracker in context, using HDFS as file system. To give a full picture, the HDFS processes, NameNode and DataNode, would have to be included. Just as with HDFS, the JobTracker and the TaskTacker will be subsequently dissected.

JobTracker

The JobTracker behaves in a very similar manner to the NameNode, but in this case it handles the scheduling part of job execution. A regular workflow starts with a client submitting both the algorithm and data to the cluster. The client is then returned an identification for the new job. The Jobtracker, when it had determined that the job should start, would initialize

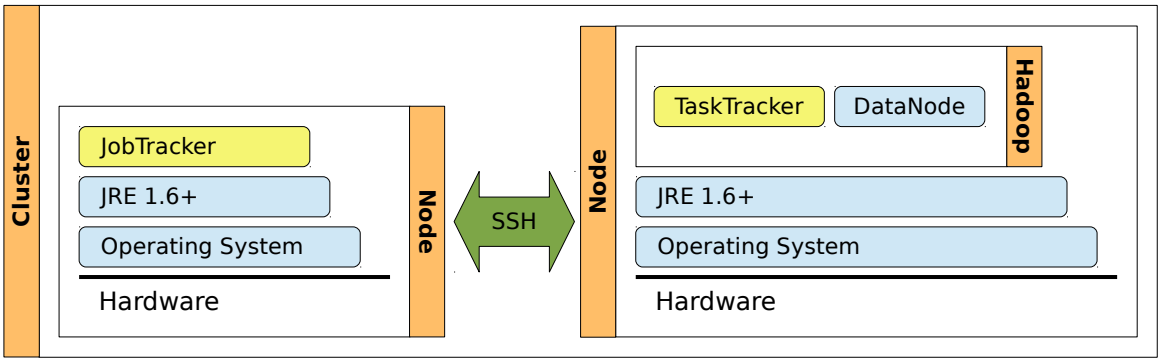

Figure 5.6: Hadoop MapReduce with HDFS

the job sending out the code required to complete the execution to the TaskTrackers. The TaskTrackers would be sent, each, a subset of the work to be completed for the job: a task in MapReduce nomenclature.

Distributing the work among TaskTrackers is done following the principle of maximum locality, i.e. making TaskTrackers process the parts of the job that refer to data stored closer to them. Enforcing this policy limits the amount of traffic flowing over the network and, at the same time, shortens data access time.

Running in an environment prone to error, the JobTracker must define a method to deal with a myriad of failures with limited impact on job throughput. To that end, the JobTracker maintains a list detailing task status. This list is updated by the JobTracker process by continually polling the TaskTrackers for updates on task status. If a TaskTracker were to stop responding after multiple tries, the JobTracker would mark the task *failed* to reschedule its execution at a later time.

More difficult to deal with would be the failure of the JobTracker process. The initial approach as proposed by Google in their paper [2] is to discard unfinished jobs and try to reboot the service in an *idle node* — and online duty-free node —, as only one JobTracker per cluster was supported. In the Hadoop ecosystem, *Zookeeper* provides the means for running multiple instances of JobTrackers within the same cluster.

TaskTracker

The TaskTracker is the process that executes MapReduce algorithms on data ideally stored in the node running the process. The TaskTracker will periodically communicate with the JobTracker to report status updates. As it has been discussed above, a failing TaskTracker will be unable to repeatedly *pong* the JobTracker's *ping*, effectively rendering the task inaccessible and due to be rescheduled to an idle TaskTracker.

Chapter 6

A Private Cloud for MapReduce Applications

This chapter introduces a novel solution that combines the virtual infrastructure managed with OpenStack and the Hadoop implementation of MapReduce to conform a powerful computational tandem. *qosh*, as it has been called, will be described throughout this section moving from architecture to implementation.

6.1 Architecture

Figure 6.1 shows a high level portrait of qosh's execution environment. The component that acts as interface between system and user is displayed on the right end. It abstracts the inherent difficulty in configuring the job execution context and in deploying the virtual cluster. Furthermore, qosh will keep track of submitted jobs, MapReduce `jar` files, input data and output results, with no need to walk the HDFS to search for data.

To streamline development, it is determined that the very first qosh version be tested on a

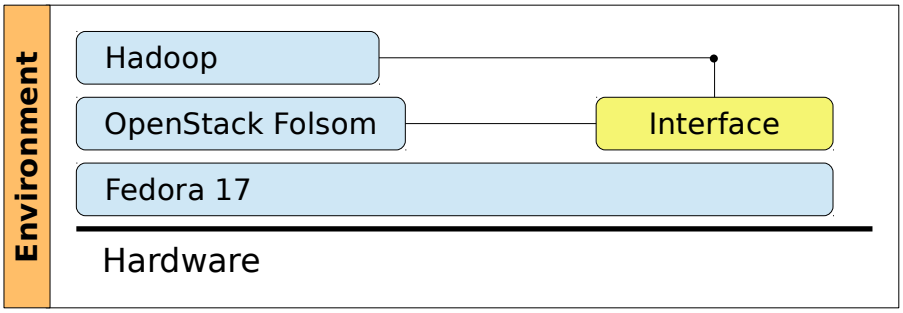

Figure 6.1: Global Architecture

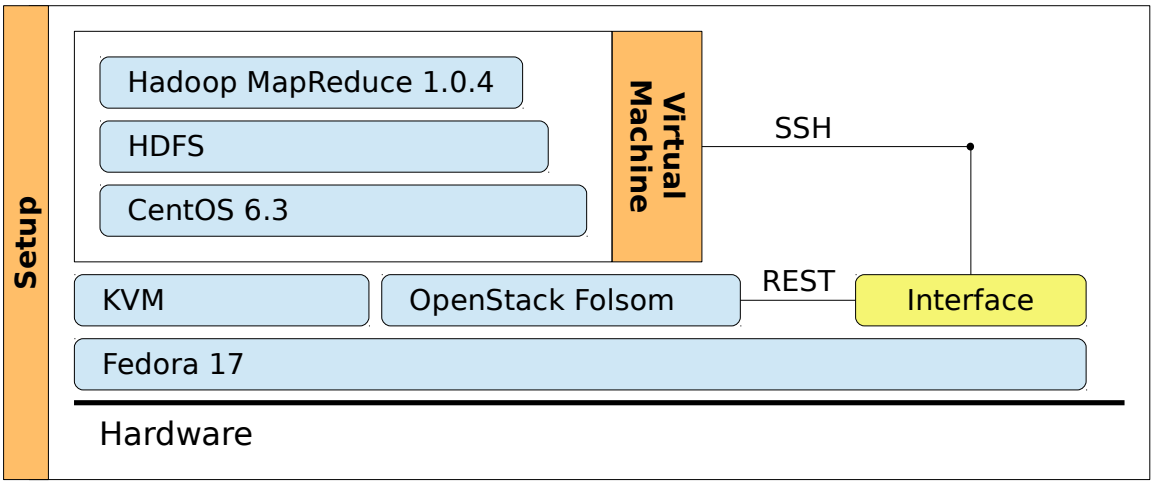

Figure 6.2: Detailed global architecture

personal computer. As shown on figure 6.1, Fedora Linux is installed atop the hardware and OpenStack Folsom set up within. qosh will draw on the infrastructure provided by the local OpenStack configuration to deploy virtual Hadoop clusters. While it would be better to make qosh more flexible allowing for remote infrastructure consumption, it would also become harder to test.

Figure 6.2 shows a more detailed view on qosh's architecture design. The VM contains a Hadoop installation ready to be put to use as soon as it was started. The VM life cycle is managed by OpenStack and its execution environment shaped by KVM, the chosen hypervisor. Besides, HDFS has been used as temporal persistence layer while the results are not send back to the controller — the same machine in this testing environment. It shall be recalled that even though HDFS is a safe data store, the Hadoop VMs are created and removed for every workflow execution, effectively destroying HDFS data at the end of processing each job. So, it is qosh's job to orchestrate data extraction before shutting down the virtual cluster.

Figure 6.3 shows the modular decomposition of qosh orchestration module.

Compute: Acts as client to OpenStack REST API. It handles every interaction with the cloud decoupling qosh from the particular IaaS Cloud.

Django: Is used in qosh to let users manage their MapReduce executions with ease via a web interface.

Fabric: Is the Python library included in qosh to configure the virtual cluster deployment and destruction.

6.1.1 Design Diagrams

Below are shown the design diagrams that make up the section on high level overview of the project.

Django Components

Figure 6.4 portrays modules adjacent to Django to support its operation.

Apache httpd: Relied on to manage user interaction with OpenStack Dashboard, it may also be used to with qosh web interface. qosh relies on Python's `SimpleHTTPServer` web server module to handle user requests, but `httpd` can be easily configured for real-life deployments.

memcached: Is employed to cache web pages in order to speedup load times.

MySQL: Chosen relational DBMS to store job meta-data.

Use Cases Diagram

Figure 6.5 displays the set of use cases that have been considered for qosh. It reflects the five fundamental agents comprising the system.

Machine State Diagram

Figure 6.6 presents a summary on the navigation flow across the web interface. An example interaction is subsequently described.

Initially, the user is presented the *Login* page so that he/she could log into qosh. If the supplied credentials were cleared — the user must be previously registered in Keystone as

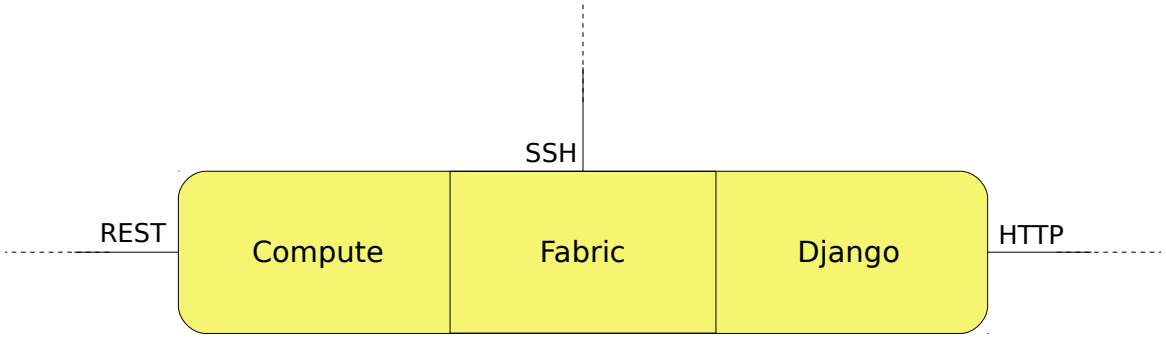

Figure 6.3: Core qosh modular decomposition

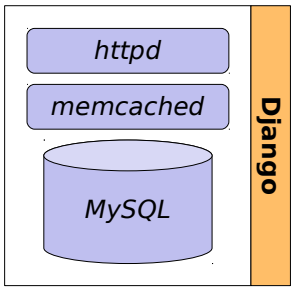

Figure 6.4: Django setup

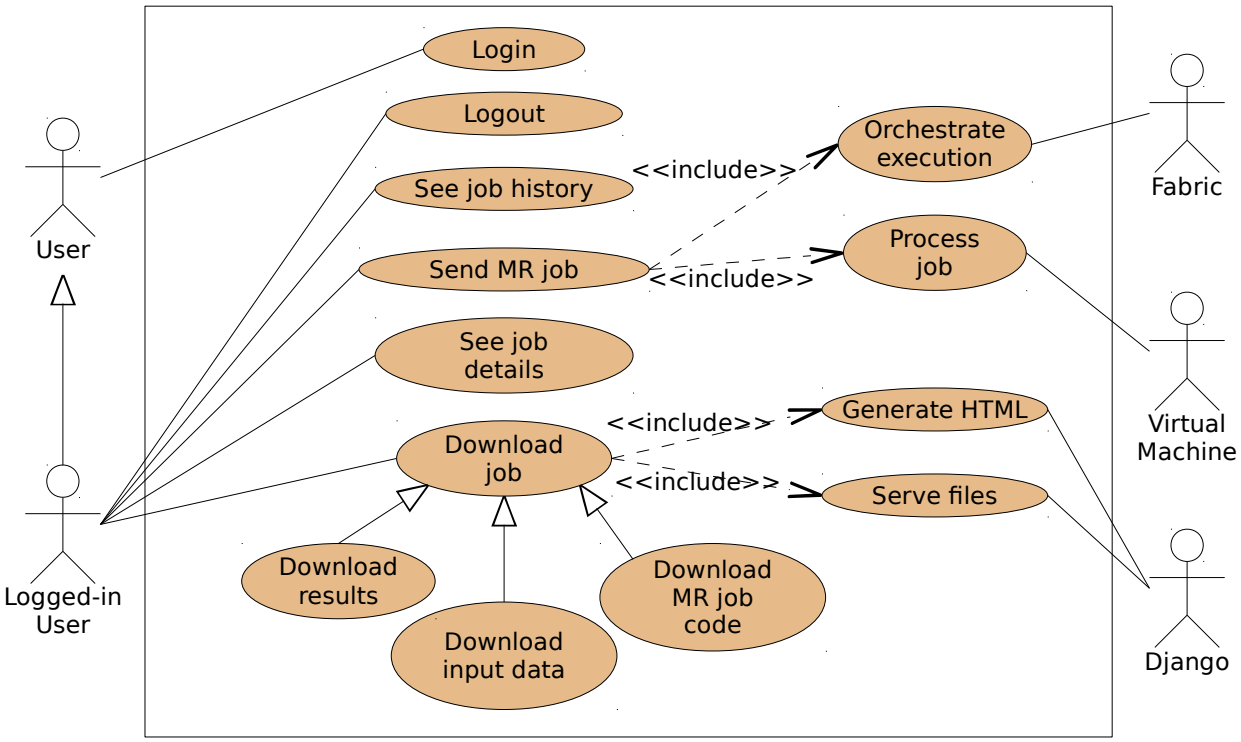

Figure 6.5: Use Cases Diagram

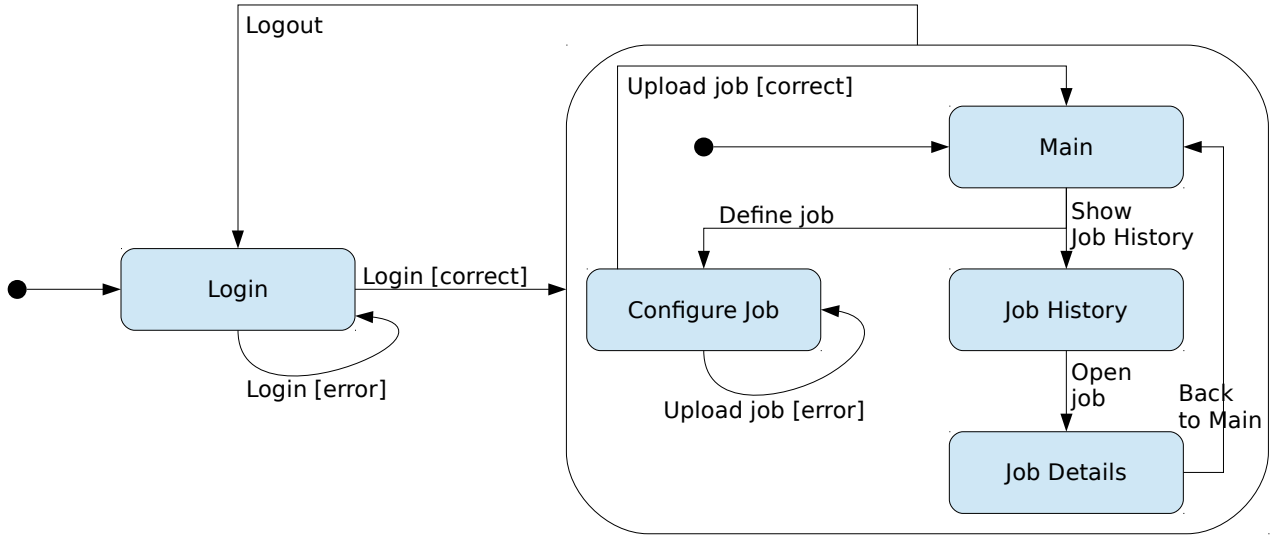

Figure 6.6: Web interface transitions

access is handled by OpenStack —, the *Main* page would be shown. From there the user may *Configure Job* or go over the *Job History* to get *Job Details*.

Class Diagram — Compute Module

Figure 6.7 shows a small Class Diagram describing how the REST client is related to Fabric and Python.

json: Parses structured JSON-formated data and permits its manipulation. OpenStack REST API understands both XML and JSON.

Exception: Python class representing a generic exception.

ServiceError: Extends `Exception` to notify and handle any error that may be raised during execution. Its objects have two member fields: an HTTP error code and an error description.

Environment: Contains the set of global options for configuring the virtual deployment.

httplib: The Python module containing functions, classes and helpers toward interacting with HTTP connections. qosh banks on it to help consume the OpenStack REST service.

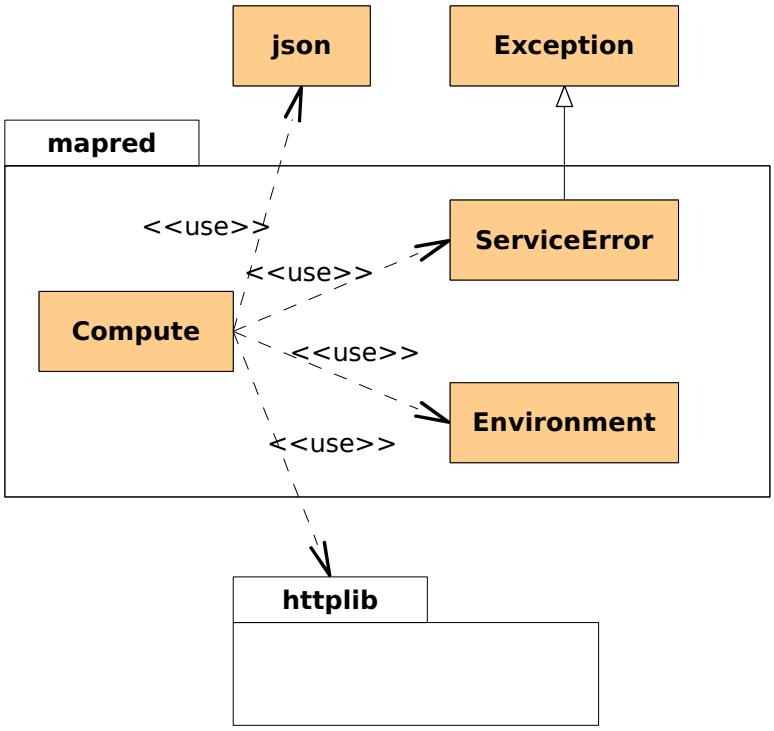

Figure 6.7: Class Diagram — Compute module (I)

6.2 Implementation

All of the three modules conforming qosh's core have been written in Python. The tests of the first version have been run over Fabric 1.4.3, Django 1.4.2 and Python 2.7.3. The configuration inside the VM is carried out by three *Bash-Scripts* in three different moments in the VM life-cycle: before bringing the network up, after having completed the boot sequence and before shutdown. They will be detailed in section 6.2.1.

6.2.1 Hadoop Virtual Machine

The Hadoop virtual machine is a vital asset for qosh; in the end it is Hadoop that executes user workflows. The versions configured are Hadoop 1.0.4 and Oracle JRE 1.7. Installing and tunning every service in the VM has not been a complex process, whereas it has been long and interesting enough, for it is potentially reusable to create other customized VMs, to list every step followed to complete the setup.

- In the local development machine the *Virtual Machine Manager* (package `virt-manager`) was installed with `yum`. It automatically installed dependent libraries like `libvirt`, the Kernel Virtual Machine hypervisor module (KVM) and a wrapper to handle it

(`qemu-kvm`).

- Through the Virtual Machine Manager, a VM with 1 GB of RAM, 4 GB for the drive image within a `qcow2` container and both APIC and ACPI was created anew.
- A minimal network install of CentOS 6.3 was carried through. *Basic Server* was chosen as initial package set on a single ext4 partition — no swap partition — without LVM.
- After the initial boot sequence, the distribution was updated to the latest *stable* version available.
- The Oracle JRE 1.7 and Hadoop 1.0.4 were downloaded from official sources and subsequently installed on the VM.
- A new user (*hduser*) was registered and set *hadoop* as its primary group. As a result, the permission of the files related to Hadoop — configuration files and scripts —- were updated to allow this new user to launch and kill MapReduce jobs.
- `sshd` was tuned to disable superuser connections and user/pass authentication method; only ssh tunnels cleared by keypairs would be permitted.
- Three scripts — similar to Ubuntu `cloud-init` scripts — were written to `/etc/init.d/cloud-*` to customize part of the VM behavior. An important feature of them being they manage injecting the public part of the keypair used for authentication into the VM file system — in `/home/hduser/.ssh/authorized_keys` — when it boots, so the holder of the private part can connect.
- The `yum groups` utility was applied to remove unused services like the *X-server*.
- From the local machine, the VM was terminated and its drive image was mounted with `qemu-nbd` to remove logs and user history from the VM. Then, a 5 GB zeroed-file was created with `dd` inside the VM file system, failing, for there was not enough free space to accommodate 5 GB within a 4 GB image, while filling the remaining space up with zeros. Just after that, `qemu-img` was invoked to compress the image into a new file.
- Next, `fdisk` was used locally to observe the VM file system block size and initial block number of the Hadoop partition — the only partition. By multiplying both values the partition offset was obtained, allowing for the extraction of the Hadoop partition using `qemu-nbd` offset-mounting capabilities and `dd` to dump, again, to a new file.
- Finally, both the *initram* and the kernel image were copied out from the VM to the local file system.

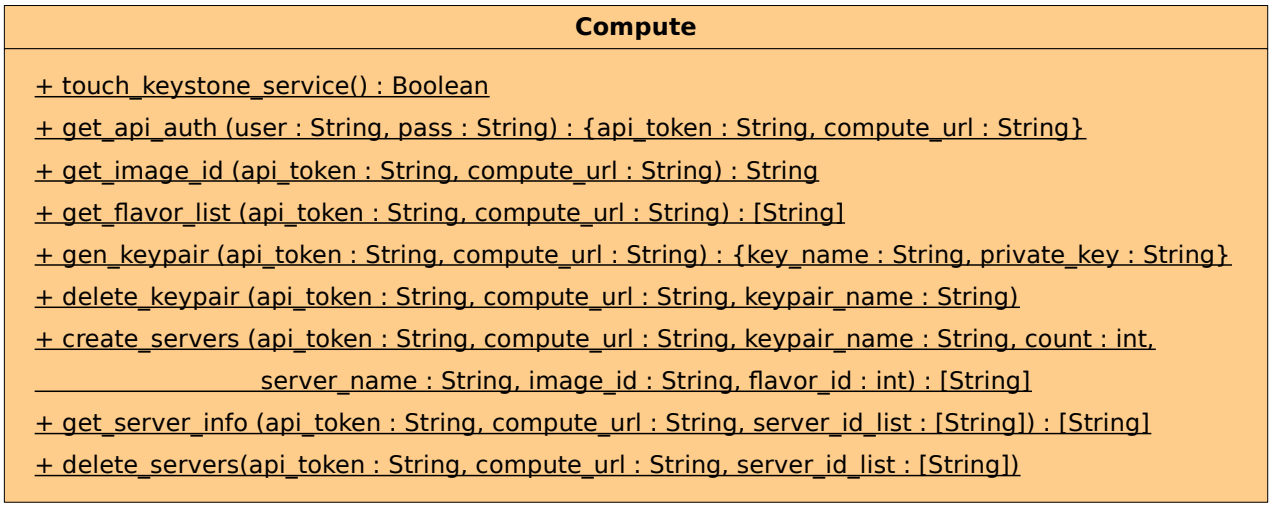

Figure 6.8: Class Diagram — Compute module (II)

6.2.2 Low Level Diagrams

Next comes the diagram set that exposes those features closer to qosh implementation.

Class Diagram — Compute Module

Figure 6.8 expands the *Compute* module details on figure 6.7. Notice that types on function signatures have been borrowed from Python syntax on dictionaries and lists for brevity.

List of type T: `[T]`

Dictionary: `{<Key1> : <T1>, <Key2> : <T2> ... }`

Class Diagram — Django and Fabric

Figure 6.9 exposes the relationship between the most important Django and Fabric modules.

Views: Is a delegate class implementing the behavior on each *view*. It interacts with qosh Compute module to communicate with the cloud.

Process: Is a `multiprocessing.Process` wrapper class encapsulating the logic to create processes that will handle Hadoop workflows execution collaborating with `Compute` and `Fabric` modules.

OpenStackBackend: Is a small class that manages user logins. It is plugged into Django authenticating pipeline to delegate the clearing of user credentials to Keystone. In so

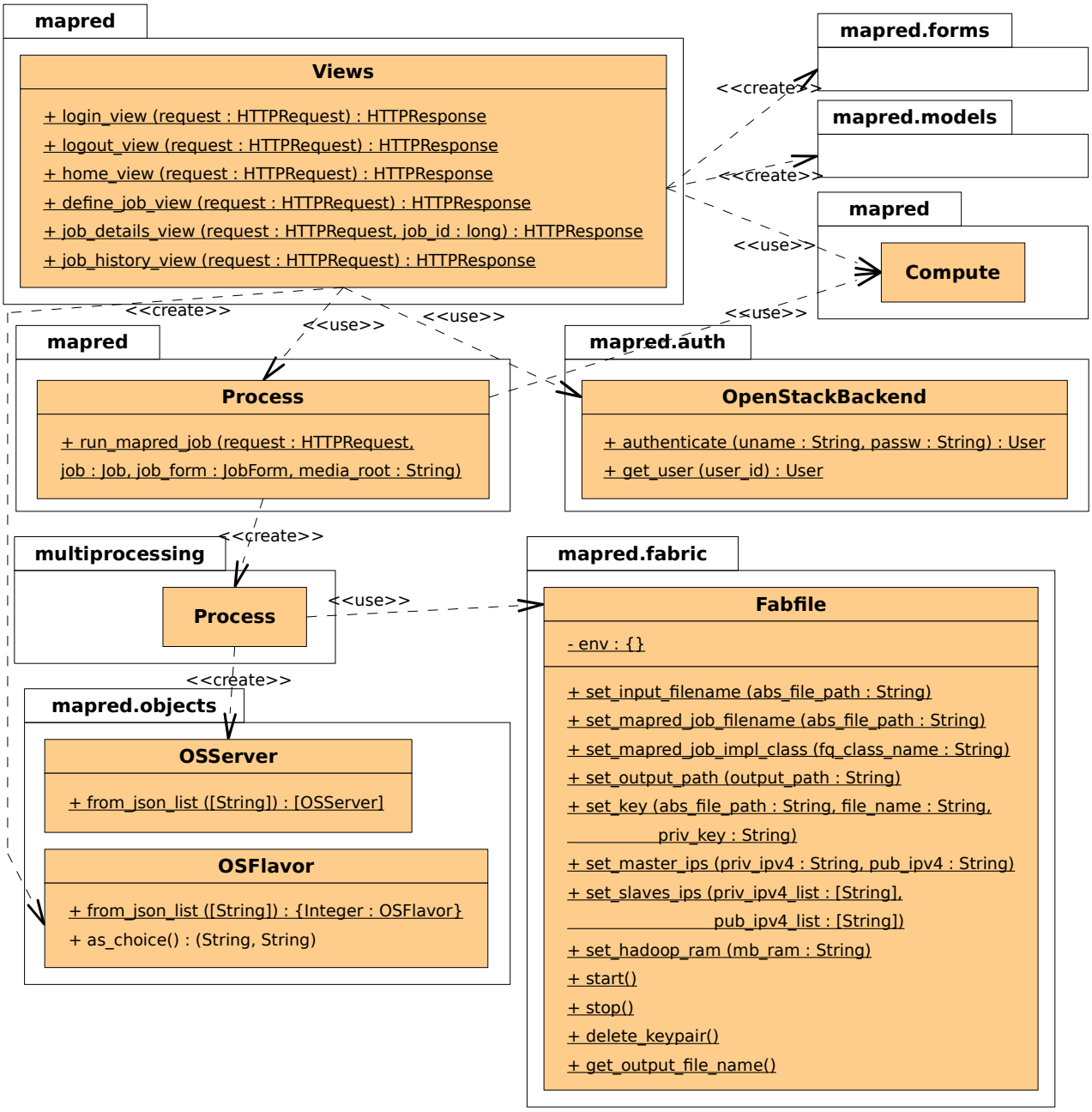

Figure 6.9: Class Diagram — Django and Fabric

doing, user access details will remain securely stored with Keystone with no need to have them duplicately stored elsewhere.

OSServer y OSFlavor: Are a kind of *Transfer Object*s holding a subset of the outputted JSON data returned from the invocation of the OpenStack REST service.

Class Diagram — Django Objects

Figure 6.10 details the relationships among the set of classes implementing the interaction with the web interface as provided by Django.

django.db.models: The objects defined therein model the business logic of our application. Django supplies a large set of templates that may be extended and/or composed to implement the particularities of different applications.

Model: Is the base class of the object model supporting the business logic. Every object within the *model layer* of our application extends this class — i.e. `Job` and `Server`. Django object-relational mapper automatically translates this object model into a relational model as well as object-based queries into SQL ones.

Fields: Hold the information related to object model classes helping Django persist the properties of the objects in a relational data base.

Job: Holds the meta-data of a Hadoop MapReduce workflow. MySQL will store these data persistently, meanwhile the file system in the Cloud Controller will actually store the information associated with the execution — the jar file with the implementing MapReduce algorithm, inputs and outputs.

Server: Contains the operating features of each virtual machine that had been part of a Hadoop job.

User: Is the Transfer Object that is used to convey user-related information from the authorization back-end to the web interface.

Class Diagram — Django Forms

Figura 6.11 shows the class breakdown supporting the job configuration-holding forms of the web interface. `django.forms` package is laid out like the `django.db.models` package discussed above, being in this case `Form`, together with any `Field` required, the extensible class Django provides to customize user forms.

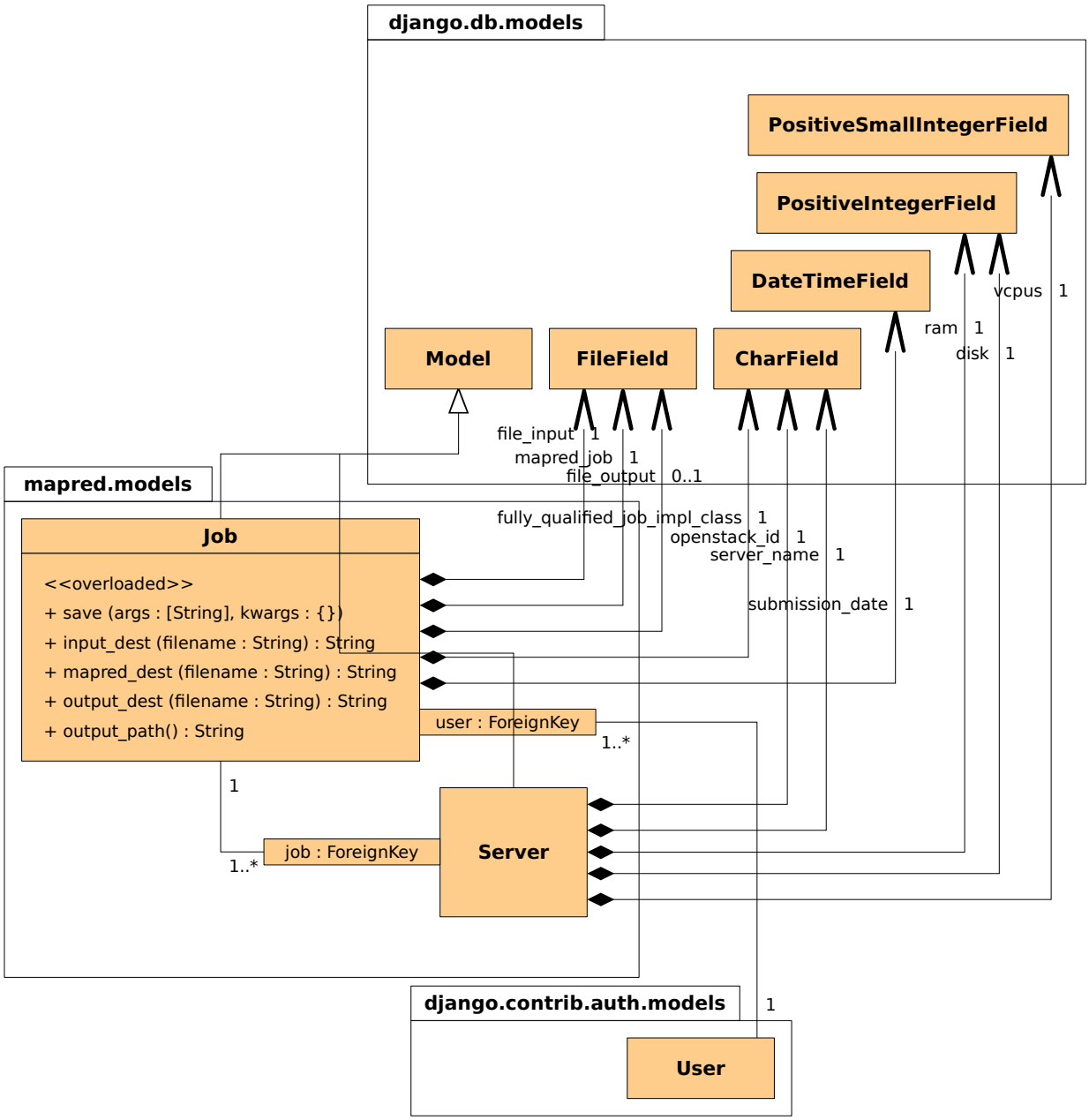

Figure 6.10: Class Diagram — Django Objects

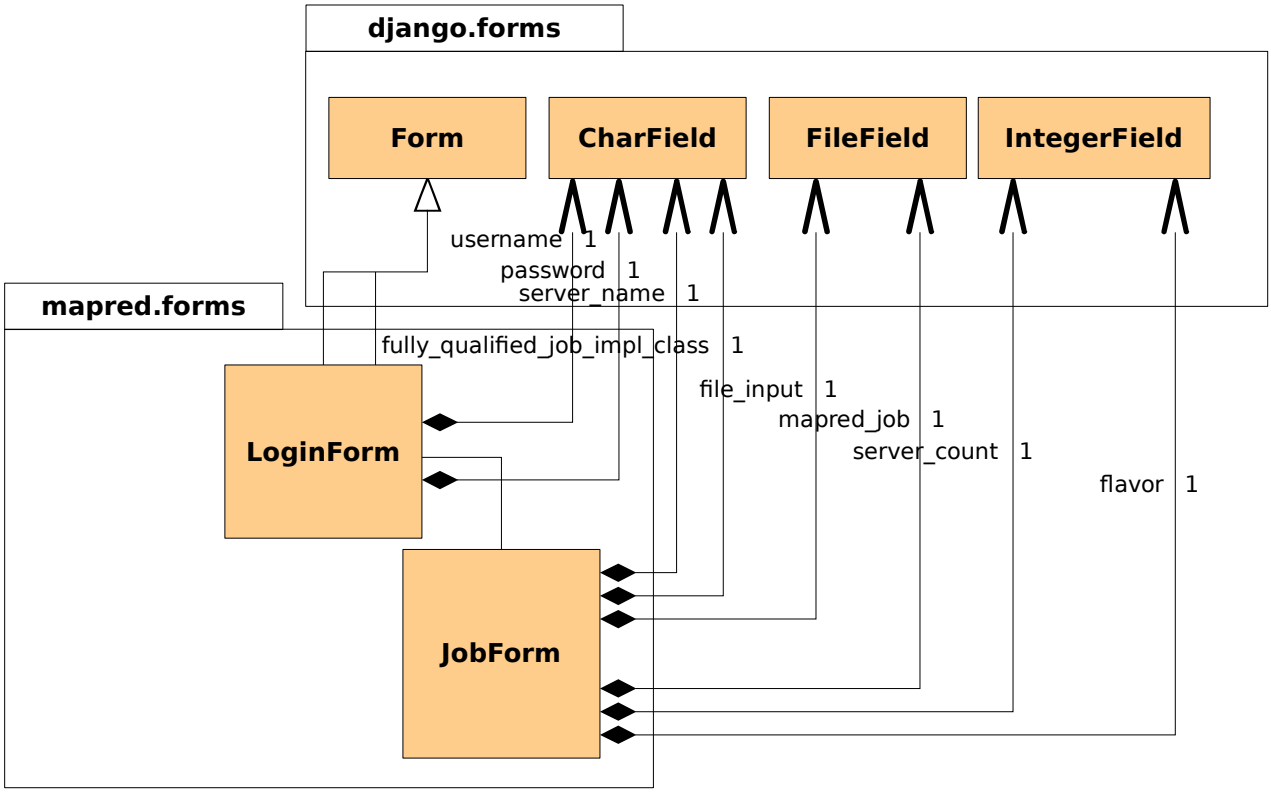

Figure 6.11: Class Diagram — Django Forms

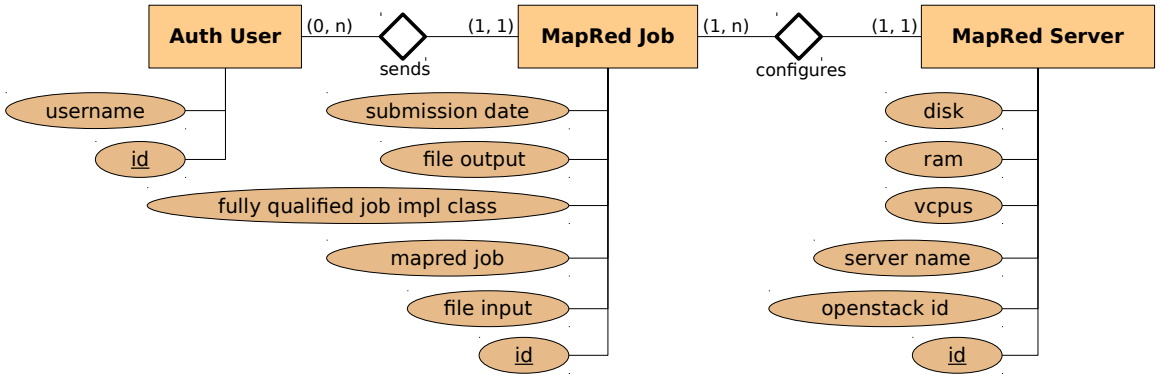

Figure 6.12: Entity-Relationship Diagram

LoginForm: Is the user sign in form. Made up of two character fields holding the user name and password.

JobForm: As its name suggests, is the form that allows for defining the workflow configuration. It is formed up by a set of `Fields` containing the following information: the virtual server name prefix, the qualified name of the Java class implementing the Mapper and Reducer, the input file, the `jar` package, the number of virtual servers to be deployed and their computational *flavor*.

Entity-Relationship Diagram

It has been briefly pointed out that Django can automatically build the data base logical model derived from the object schema written by the developer. And as expected, *CRUD* operations on the DB are also managed by Django. Figure 6.12 shows the actual *Entity-Relationship* diagram of the schema that is contained in MySQL.

Sequence Diagrams

Figures 6.13 y 6.14 present two Sequence Diagams. They reflect the most interesting subset of messages that are interchanged between the different execution-participating entities. The sequences suppose that no errors raise during the interaction, that the user has previously signed in, that every job defining information is typed correctly and that the quota lets the deployment of the virtual cluster be carried through. Figure 6.13 pictures the full interaction. Figure 6.14 details the collaboration between classes after message *#24* in figure 6.13.

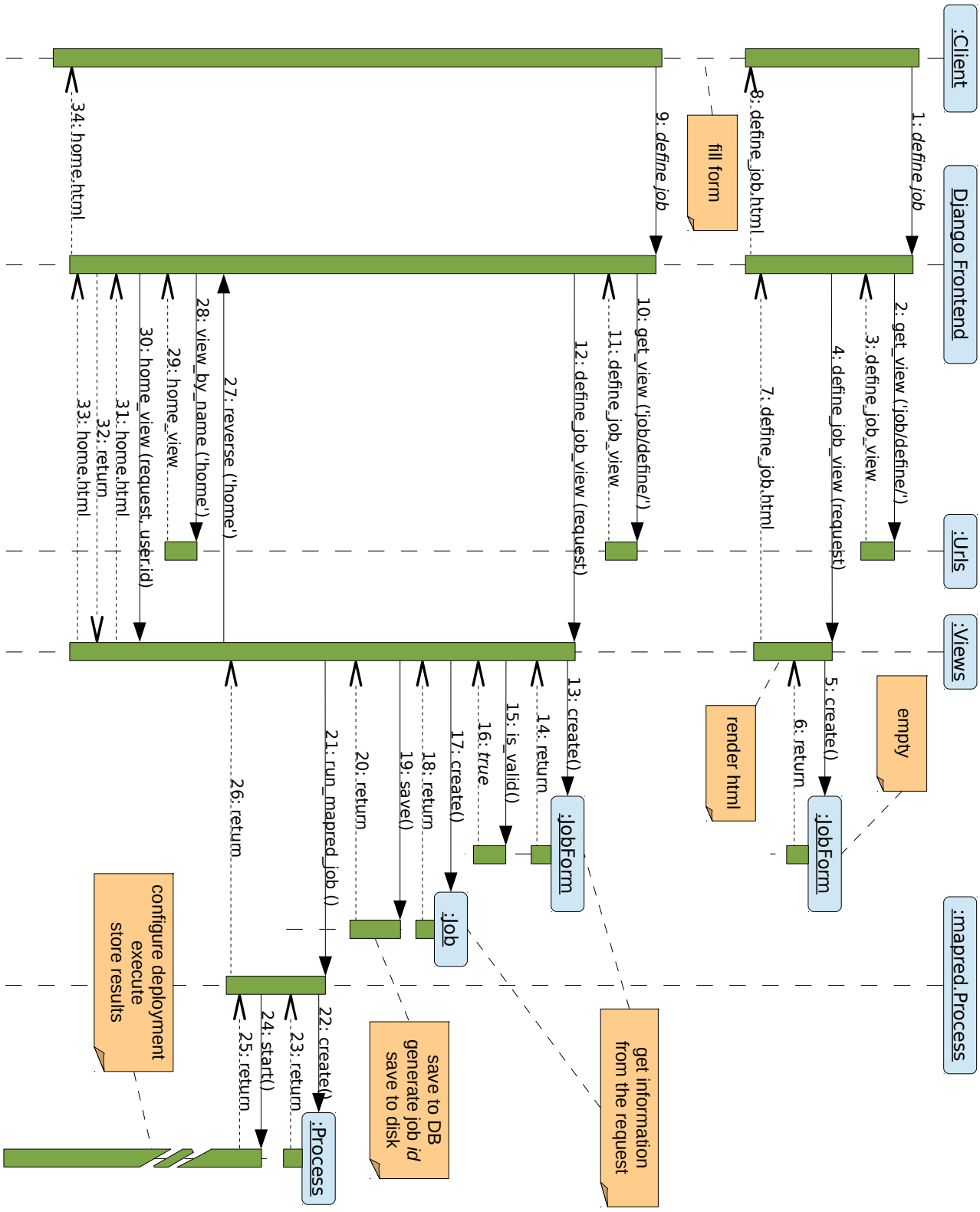

Figure 6.13: Sequence Diagram (I)

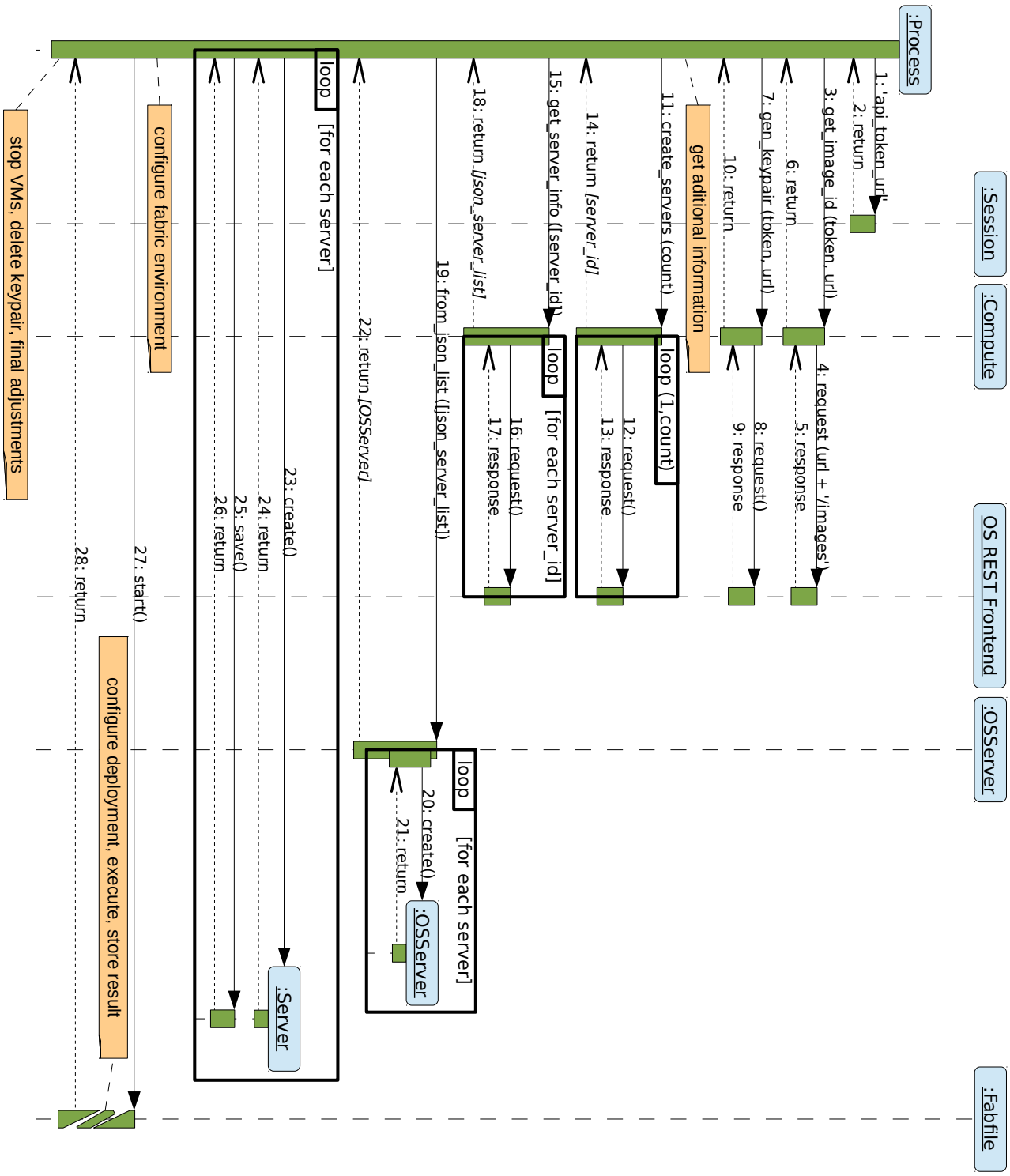

Figure 6.14: Sequence Diagram (II)

Chapter 7

Performance Analysis

In this chapter qosh performance will be evaluated deployed on a real cluster. The execution environment will be described first and the metrics collected will be analyzed and put in context thereafter.

7.1 Testing Environment

Figure 7.1 shows how the cluster is configured. The leftmost part of the figure shows the Cloud Controller; the rightmost, the Cloud Node. The Cloud Controller has been installed the full OpenStack Folsom package — except for Cinder, Swift and Quantum as they will not used —, plus MySQL, Fabric and Qpid as message broker. In the Cloud Node only the bare minimum to support VM execution has been installed — OpenStack Compute and KVM. Other functional requirements of the Cloud Node, like access control or VM image download, are delegated to the Cloud Controller via Qpid.

Regarding the physical layer, both nodes posses the same hardware configuration: Octa core Intel Xeon processor @ 3.2 GHz with VT-x, 8 GB of RAM, 200 GB SATA 3 Gb/s 7200

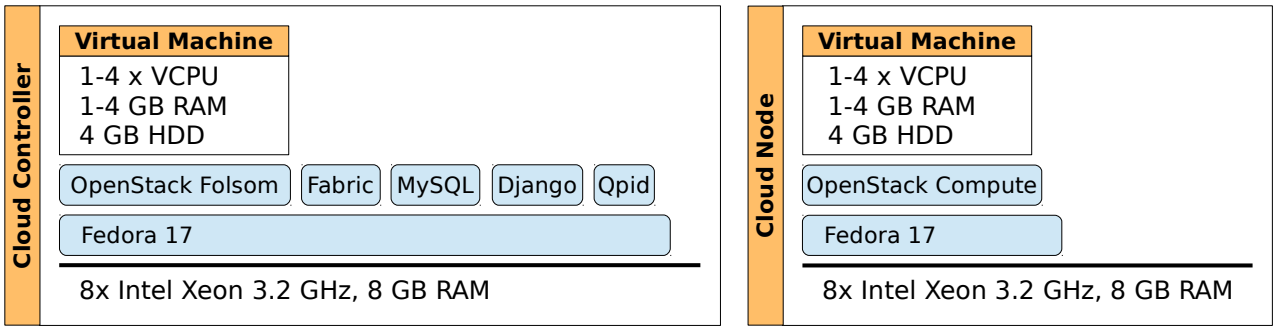

Figure 7.1: Deployment morphology

RPM and Gigabit networking interface.

The Hadoop virtual image, whose creation procedure is described in section 6.2.1, will be used in the VMs to execute MapReduce workflows. Figure 7.1 shows the Hadoop VM in context. This instance will be provided with 1 to 4 GB of RAM and the same number of VCPUs to assess qosh's scaling. The deployment has been configured to allow Hadoop to address all the memory in the VM — except for those addresses in use by the OS and the JVM.

7.2 Testing Methodology

This section contains both in and out scalability analysis and a study on qosh behavior on increasing input sizes. To evaluate how qosh scales, a MapReduce workflow large enough to stress every instance in the virtual cluster, will be fed to Hadoop; the workflow will count the words in 62.5 MB of plain text.

To actually assess horizontal scalability, the size of the virtual cluster will progressively be doubled starting from one instance up to four. To test vertical scalability, two VMs will be initially spawned with 1 GB of RAM each, to have their RAM and VCPU count doubled on each test case until reaching 4 GB of RAM and 4 VCPUs. Finally, to evaluate the running time tendency, the input text size will be doubled from the original 62.5 MB to 250 MB.

As Glance and Compute cooperate to cache images, the time required to start an instance will be longer the first time it launches, effectively skewing results. To get rid of those divergences, the instances on each flavor will be *warmed* by starting and destroying them once before taking measures.

A priori, the expected variance of the timings between executions is small enough to consider *ten* to be a representative number of runs for each test case — this hypothesis will be validated *a posteriori* on analyzing the results. To measure timings, the code will include a set of time marks that will help take apart the following times:

Deploying: Time elapsed since the virtual cluster creation request is send until every instance is accessible.

Configuring: Time required to establish the Hadoop execution environment. It comprises the time to configure the virtual Hadoop cluster plus the time to distribute input data onto HDFS.

MapReducing: Time dedicated by Hadoop to execute the *wordcount* workflow.

Cleaning: Time required to wipe out the execution environment. It does not include the time it takes OpenStack to completely remove the instances.

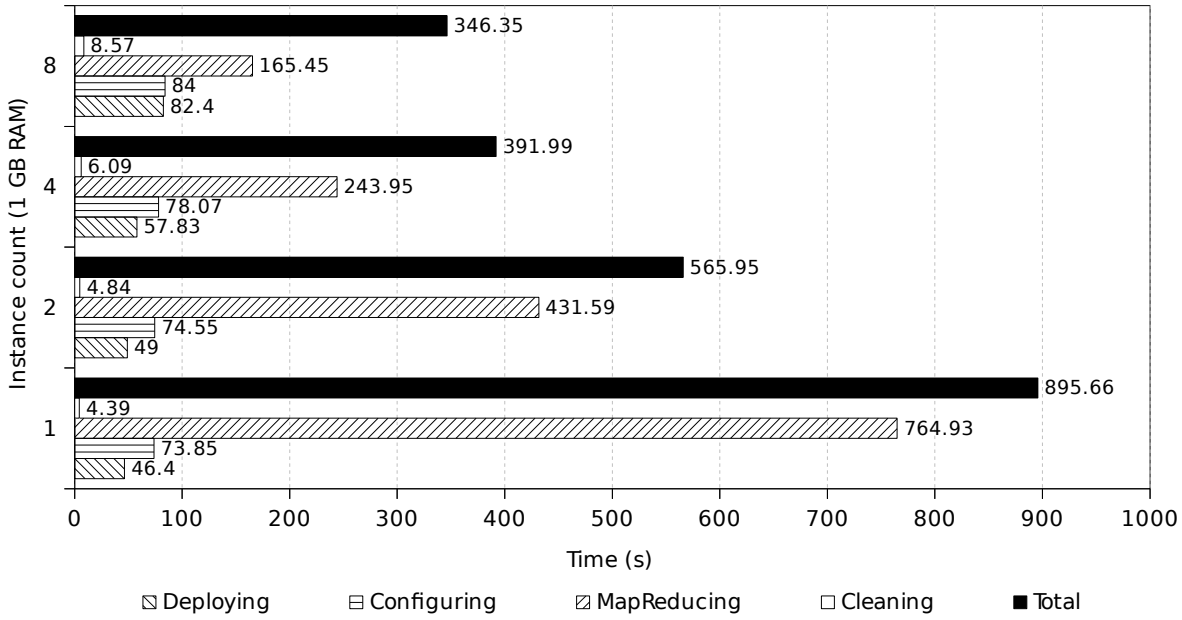

Figure 7.2: Scaling out

Total: Time required by the whole process to complete. It is not the result of adding the previous four times.

It should be noted that the timings shown are the averaged results of the ten executions in each case.

7.3 Analyzing the Results

Figure 7.2 shows the evolution of the different timings as the instance count increases from one to eight. It may be observed that deployment, configuring and cleaning time increase with the number of instances deployed, meanwhile processing time is reduced. I.e, the time required for a virtual cluster to deploy, configure and delete every instance depends *only* on its size. On the other hand, MapReducing time — and thus total time — is also bound to input size. Figure 7.3 presents the tendency of the five timings as the cluster is scaled in from 1 GB of VCPU and RAM to 4 each.

It should be highlighted from figure 7.3 that deploying and cleaning times are approximately flat even though the instances are made larger. I.e., the difficulty to deploy or delete a virtual cluster in our environment depends exclusively on the number of instances or virtual nodes — conclusion we had drawn earlier. Configuring times in this case are reduced, for the instances

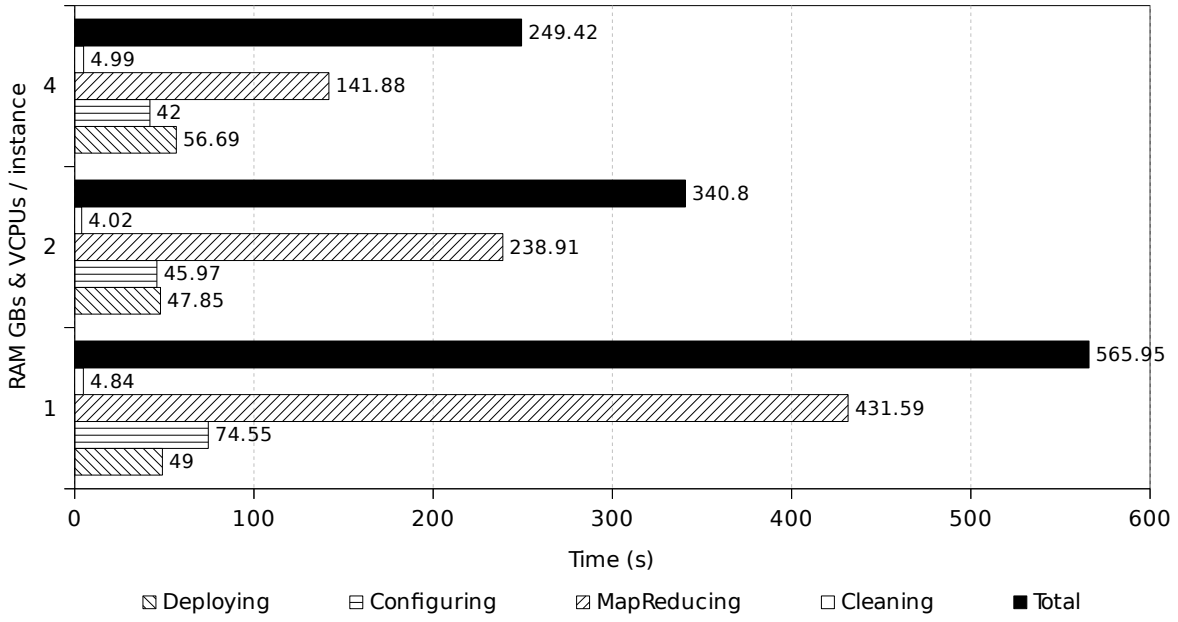

Figure 7.3: Scaling in

are more capable computationally — while testing horizontal scaling revealed that configuring time increased proportionally with the cluster size.

In light of the results, it can be concluded that qosh behaves more efficiently on vertical scaling versus horizontal scaling. This fact is specially evident when contrasting the extreme testing cases from figures 7.2 and 7.3: eight instances, 1 GB of RAM and 1 VCPU each and 4 GB of RAM and 4 VCPUs respectively. Both test cases deploy a virtual cluster with 8 VCPUs and 8 GB of RAM over the physical infrastructure. Yet, total execution time is almost *28%* smaller with two larger instances.

Figure 7.4 shows the tendency on execution times when increasing the input size. As expected, MapReducing and total times increase with input size but in a smaller proportion. Deploying and cleaning times stay almost constant on each case, while configuring time increases slightly as it covers decompressing and distributing the input files over the cluster.

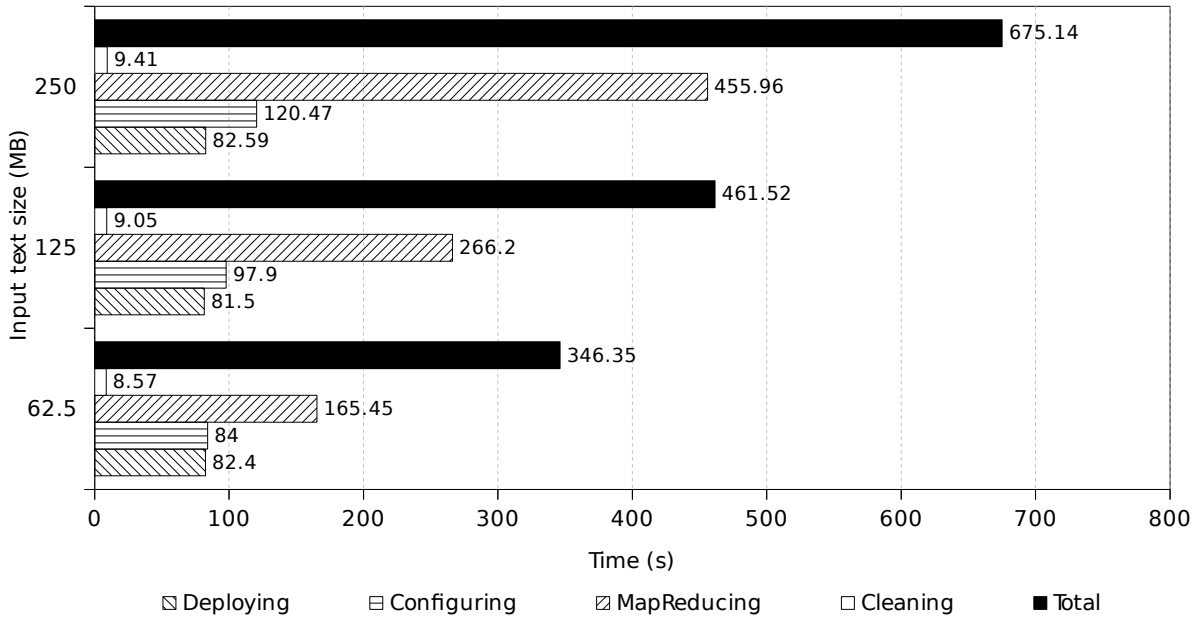

Figure 7.4: Input size versus execution time

Chapter 8

Related Works

The present chapter describes other particular solutions that explore executing MapReduce applications over on-demand infrastructure provisions. Each one's characteristics will be briefly explained and contrasted against qosh's.

8.1 Amazon Elastic MapReduce

Amazon Elastic MapReduce [29] exposes a web service interface that allows a user to send MapReduce execution requests. EMR is supported internally by Amazon EC2 to provision, on-demand, the infrastructure required by the MapReduce workflow. Thus, an EMR user will not have to worry about provisioning, creating, configuring and destroying the virtual clusters that would support execution. This convenient transparency to the user comes with a series of relatively important limitations:

Underlying infrastructure restriction: Being as it is a service owned by Amazon Inc., it was expected that usage of computational resources out of their reach were limited; and so it is: EMR users will not be able to couple their virtual instances to other clouds that may incur in lower exploitation costs. qosh, as it has been shown, separates and exposes responsibilities in a way that using a different cloud, requires only adapting a new Compute module to the particular REST API of the cloud that would be used.

Installation restriction: It is not possible to deploy a custom-built VM to execute MapReduce applications in EMR. qosh will handle workflows driving a Hadoop VM with a loose coupling with the deploying subsystem (Fabric) with a doble end:

- To ease VM customizations and upgrades (Kernel, Hadoop, JRE, etc.).
- To give the possibility to create custom VMs from scratch.

Information restriction: Some users are under non-disclosure agreements that render impossible sharing data with third parties like Amazon, effectively leaving those services out of the equation. qosh's open source nature makes it easy for a developer to adapt qosh to its particular functional requirements with no sharing of private data.

EMR node typology is described next. EMR clusters present three kinds of nodes:

Master: this node, unique per EMR cluster, executes both a NameNode and a JobTracker.

Core: This kind of nodes store data and process job tasks. They are basically comprised of a DataNode and a TaskTracker.

Task: they only run TaskTracker processes.

qosh deploys clusters with a single hybrid master-worker node — with NameNode, JobTracker, DataNode and TaskTracker — being the rest worker nodes — with DataNode and TaskTracker only. This deployment configuration could be easily altered to fit into diverse use cases by rewriting the `mapred.fabric.fabfile` module.

Regarding the execution flow, EMR allows for keeping the instances alive when they had finished their scheduled processing to avoid the computational overhead of their spawning. qosh's default behavior is to destroy instances upon completion of each workflow. As expected, defaults may be easily overridden.

EMR draws on Amazon S3 to store input data to Hadoop as well as to write final results. Intermediate information is stored within the virtual instances and is destroyed as they are. qosh uses the file system of the Controller Node to store I/O data from the virtual cluster, and, like EMR, employs the virtual instances' file system to save intermediate data. qosh may also be plugged a different repository to store final results. To that end, a new back-end module would have to be written for Django to correctly locate data and update the DB to point elsewhere.

8.2 Resilin

Resilin's [3] main intention is to improve EMR by trying to overcome its limitations. The development team has written an API (see figure 8.1) capable of receiving EMR-like requests that will translate into two kinds of actions:

- An interaction flow with an IaaS Cloud to create and destroy instances (what qosh does in its Compute module).

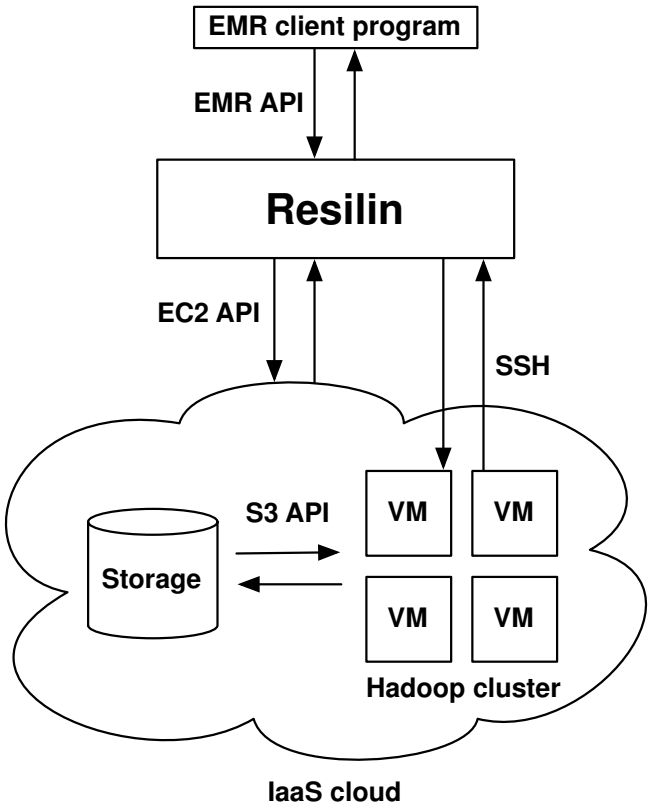

Figure 8.1: Resilin architecture. Source: [3]

- An SSH connection to the virtual Hadoop cluster to configure and execute tasks (what qosh does with its Fabric module).

Information storage and I/O is delegated upon an S3 compatibility adaptor supported both by Nimbus and Eucalyptus — with *Cumulus* and *Walrus* respectively.

Besides, and maybe the most interesting Resilin trait, its developers experiment with the possibility to execute MapRecude workflows atop infrastructure provided by different clouds. Though it may not be the general approach to provision private infrastructure to virtual deployments due to the inherent heterogeneity among different IaaS frameworks, it might prove useful when the infrastructure has to be drawn upon from public clouds. In the abstract, the idea has been implemented bridging deployments on different clouds by exposing instances on one cloud to the other cloud Master Node as if they were part of the same virtual cluster. qosh does not support hybrid cloud deployments. Yet, it cloud be implemented without much ado by modifying the Compute module accordingly.

Another salient feature of Resilin's is the capacity to grow or shrink virtual clusters when the processing had already started. New instances will be assigned tasks as soon as they are registered in the Master process. qosh does not support online cluster resizing.

	Resilin	qosh
Global	`Python`	`Python`
HTTP	`Twisted`	`Django`
IaaS Cloud interaction	`boto`	`Compute` (custom module)
Virtual instances interaction	`paramiko`	`Fabric`

Figure 8.2: Tool listing

The bottom line is that Resilin is the solution resembling qosh the most; delegating the reduction of MapReduce workflows on Hadoop and the virtual deployments on an IaaS Cloud EC2-compatible — all of the four implementations evaluated in 3.2 are. Lastly, figure 8.2 lists the different tools that aid supporting the functionality exposed.

8.3 Cloud MapReduce

Cloud MapReduce approaches the problem of processing MapReduce workflows over *elastic* infrastructure in a different manner. While Resilin tries to orchestrate executions by putting together every component required to carry computations through, Cloud MapReduce is backed by Amazon's EC2 services to implement the MapReduce model [2] *directly* on top of them.

Following there is a list of the most relevant characteristics of Cloud MapReduce.

Incremental scalability: Is the ability to add new nodes to the virtual cluster when job processing had already begun. New instances will fetch the global job state and will be assigned new tasks as they become online.

Symmetry and Decentralization: Is the quality by which Cloud MapReduce will deploy virtual clusters with a flat hierarchy. This feature represents the first separation from what it has been discussed hitherto, where at least one node was in charge of scheduling the execution and storage of others. In Cloud MapReduce the nodes in a virtual cluster collaborate independently from each other, choosing to process the task that would best accommodate their capabilities and that it would work best from a global job perspective. This symmetry makes it easier for a cluster to overcome failing nodes as there is no single point of failure.

Heterogeneity: Understood with a double component: On the one hand, the possible coexistence of VMs of different computational flavors in the same workflow; on the other, the ability to create instances on multiple clouds. Resilin supports both features, qosh does not out of the box.

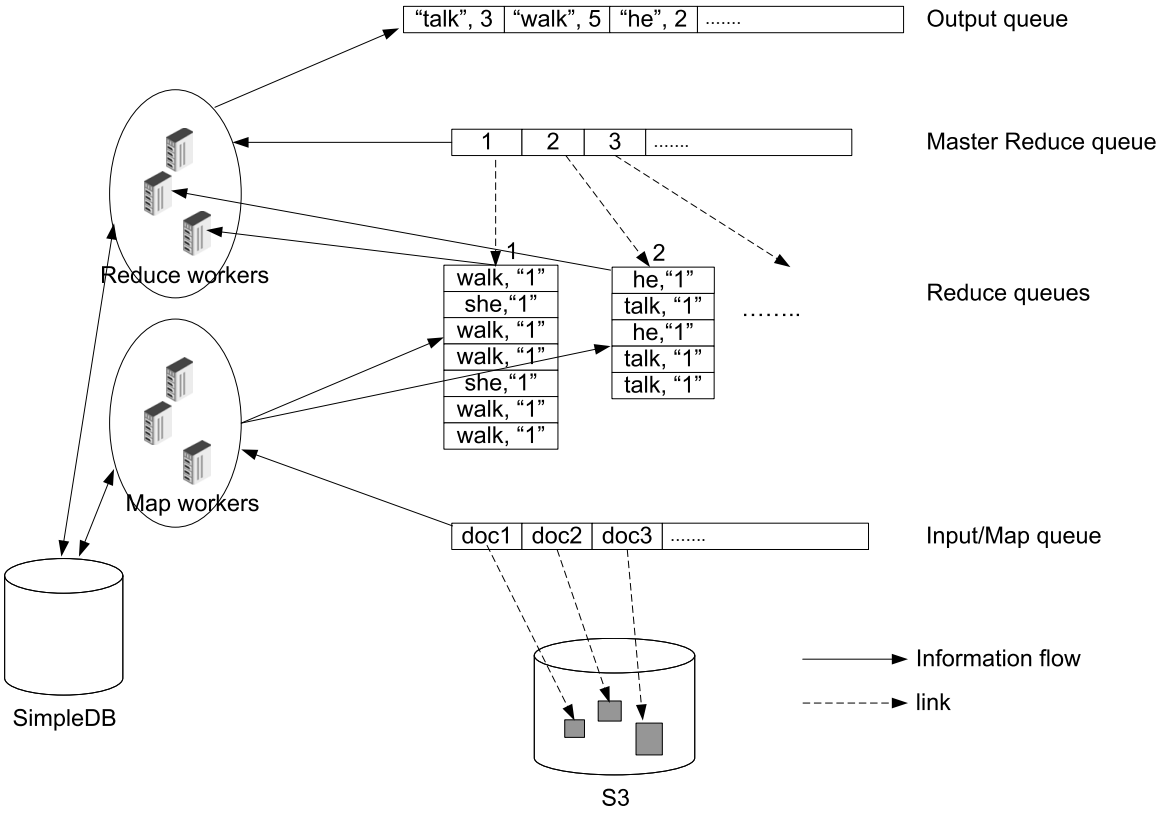

Figure 8.3: Cloud MapReduce architecture. Source: [4]

Cloud MapReduce stores I/O and intermediate data into S3. Inter-node communication is solved using Amazon queue service (*SQS* or *Simple Queue Service*). Meta-data is saved in *SimpleDB*.

Lastly, figure 8.3 represents a high level snapshot of Cloud MapReduce. This figure resembles, not coincidentally, figure 2.9 which showed the typical MapReduce execution phases.

8.4 Dynamic Cloud MapReduce

Figure 8.4 shows the high level architecture of Dynamic Cloud MapReduce [5]. It can be easily seen how these high level components directly correspond to qosh's, though Dynamic Cloud MapReduce presents a series of disparities.

GUI: Dynamic Cloud MapReduce implements a RESTful web service to decouple job management from the particular interface with the user, though the main interaction with the service is through a web site plugged directly to the web service.

Chain Workflows: Dynamic Cloud MapReduce allows for chaining MapReduce executions

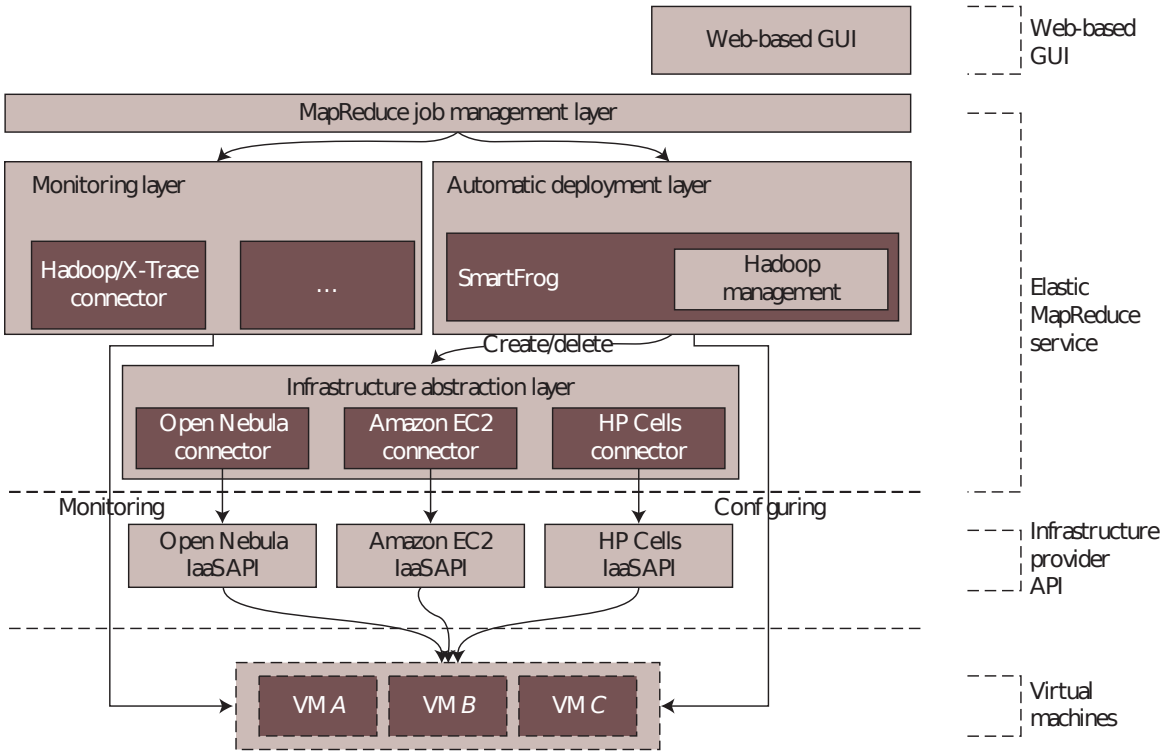

Figure 8.4: Dynamic Cloud MapReduce. Source: [5]

with no need to recreate the virtual cluster to process new jobs.

Monitoring Layer: As its name indicates, Dynamic Cloud MapReduce dedicates an independent front-end to oversee the framework status. qosh does not implement a separate layer to supervise the health of the cluster or the evolution of jobs, but this information could be easily extracted from OpenStack Dashboard or Hadoop web interface.

Automatic Deployment Layer: Being functionally equivalent to qosh's Fabric module, it controls the virtual deployment. The main difference is that virtual instances are created empty: besides the core OS only *SmartFrog* is installed. Hence, before mapping or reducing, every VM will delegate to SmartFrog the download, installation and configuration of Hadoop. While more flexible, this approach introduces an important overhead compared to qosh's preconfigured VM that, nonetheless, would become less significant as the job size increased.

Infrastructure Provider Abstraction Layer: This facade separates the REST web service from the particular IaaS Cloud in use, allowing for the definition of adapters to deal with each one. qosh does not abstract an interface to implement different cloud adapters transparently, yet, rewriting the behavior of some functions in the Compute module and

accommodating the Transfer Objects to hold the different data would suffice to support any cloud.

Again, the storage back-end is a combination of HDFS and the instances' local file system but it requires the user to manually upload input data to the cluster, instead of managing inputs automatically as qosh does.

8.5 Summary

Bearing in mind all that has been discussed up to this point, it may be though that qosh is the most limited of the solutions. However, it introduces an important advantage: its utmost simplicity. What could be a subordinate matter comes to the foreground if the user is not an expert or if a quick *elastic* Hadoop test-drive is due.

No other solution presents such a simple and self-contained installation, configuration and infrastructure exploitation. They all need a larger effort from the user to start processing MapReduce workflows. In general, they require: knowing their inner workings, a preexisting deployment of an IaaS cloud, *pay-per-use* or share inputs with third parties. qosh, is the natural choice for initially small deployments or as introductory point to MapReduce and IaaS Cloud technologies. Starting from the minimum automatic deployment, the admin. could add new nodes to the cloud, update the Hadoop VM, alter the infrastructure provisioning mechanics or install another IaaS Cloud with no effort.

Conclusion

This section discusses a series of qosh's contributions and possible lines of development for the future.

Main Contributions

qosh has been designed and implemented to be the solution that will considerably reduce the complexity and cost inherent to elastic MapReduce computations. qosh automatically configures a minimum OpenStack Folsom installation before being able to drive a virtual Hadoop cluster deployed atop. As a convenient feature, qosh is also accompanied with a web interface to setup MapReduce jobs and fetch results as they finish execution.

The main qosh characteristics may be summarized as follows.

Simplicity both for installation and exploitation. The installer cannot be any easier and the web interface provides the means to define Hadoop clusters indeterminately large, send in MapReduce workflows and retrieve results directly and intuitively. The Annex A details a quick guide to complete a testing deployment.

Vertical integration of every component that allows qosh function. The installation script sets up the execution environment for every level — web, MapReduce and infrastructure — needing no interaction from the user.

High performance in the initial deployment. qosh has been devised to be simple and flexible but also performing. Other simple solutions like *DevStack*, while easy to configure and run, impose a serious penalty to performance rendering them unable to provide the required environment to develop applications on top of OpenStack — which, of course, it is not DevStack's main purpose.

Reusability of fundamental components. By having followed AWS' guidelines of letting the injected meta-data configure the instances when booting, it allows any IaaS Cloud to be used — provided it be AWS-compatible —, or seen from another perspective, it allows the default Hadoop VM to be installed directly on a real cluster, i.e. without virtualization.

Adaptability of qosh's to handle infrastructure from other clouds. Using the Compute module as starting point, a partial rewrite would suffice to support any IaaS Cloud.

Transparency of the whole. The code of any module may be openly accessed, downloaded and modified — some restrictions may apply —; including OpenStack, Django, Fabric, Hadoop, Python and qosh.

Future Development

One of qosh's weak points lies in the coupling among the main modules. It would be interesting to abstract an interface with the cloud's REST API access client, so that different delegates could implement it and adapt the messages to the particular cloud syntax, opening indirectly the possibility to hybrid cloud deployments.

Following those dynamics, some use cases like *Show History* or *Send Job*, could become objects, decouple from the web interface and be executed in an action processor object. Having unlinked those actions off the interface, it would not be hard to implement a particular REST API that would allow clients of any nature to execute MapReduce workflows. This action processor object might be designed as a thread pool that would consume object-actions fed from the interface controller, facilitating scaling out and load balancing.

Appendix A

Installation and Exploitation

To make a quick deployment in a single node and be able to run MapReduce workflows effortlessly, a *git* repository has been set up with source files, configuration scripts, and the Hadoop VM. This guide will describe the steps required to achieve a basic installation.

A.1 Quick Installation

A.1.1 System Requirements

qosh's operating requirements:

- x86_64 CPU with virtualization extensions (VT-x or AMD-V).
- 4 GB of RAM or more.
- 10 GB of HDD space.
- Fedora 17 installation.

The amount of RAM and HDD will limit the number of concurrent instances that are allowed to run — recall that each instance will be provisioned 1 GB of RAM and 4 GB of HDD on boot from the host computer.

A.1.2 Installation

Figure A.1 contains the command list that will have to be typed in a terminal to run the installation and configuration scripts.

```
$ sudo yum install -y git
$ sudo yum update -y
$ sudo reboot

$ cd <destination_folder>
$ git clone https://code.google.com/p/quick-openstacked-hadoop
(quick-openstacked-hadoop folder will be created)

$ cd quick-openstacked-hadoop
$ sudo python install.py
(OpenStack Folsom, Hadoop VM, Fabric and Django will be installed)

$ python configure.py
(Django will be adjusted)
```

Figure A.1: Command sequence (I)

A.1.3 Test-driving the Setup

If the commands in the previous section have succeeded, OpenStack and Django should be correctly installed. To assert this, open `http://localhost/dashboard` in a web browser. After a few seconds there should appear the welcoming Horizon splash screen that would allow to log into OpenStack — user name and password default to *udc*. Once logged in, a web page like the one in figure A.2 should be displayed.

At this point Django's development web server may be started to directly interact with qosh and check whether its installation succeeded — figure A.3 shows the command sequence to start the server. Again, a web browser shall be used to open `http://localhost:8000/albaproject/mapred`; then, qosh's log in interface should load. User name and password are *udc* again — recall that user access is delegated entirely to OpenStack Keystone. Figure A.3 lists the commands that should be typed in a terminal.

A.1.4 Testing a Workflow for Hadoop

Finally, to ensure every component is completely installed and configured, a real MapReduce workflow should be run in qosh. Input files for this test case might be downloaded from Google Code repository at `https://code.google.com/p/quick-openstacked-hadoop/downloads/list`. There should be the files `just_imagine.tar.gz` and `bigtxt.tar.gz`, zipped files con-

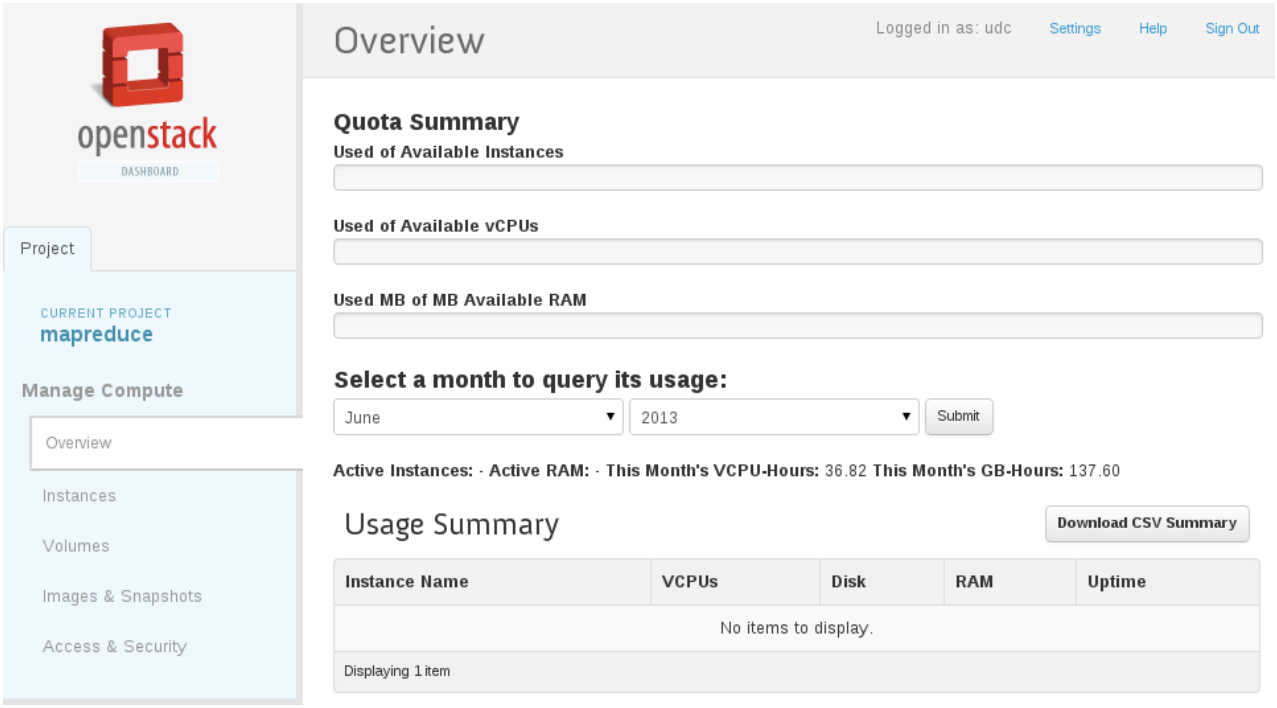

Figure A.2: OpenStack Folsom overview page

```
$ cd quick-openstacked-hadoop/Alba/albaproject
$ python manage.py runserver
```

Figure A.3: Command sequence (II)

taining plain-text files whose words will be counted, and `wordcount.jar`, a package containing the classical word count MapReduce implementation.

Just after logging in, qosh's main page should load on screen. From there, Define Job shall be clicked and the form to set up the MapReduce job should appear. The fields should be filled in in a similar way as in figure A.4 before clicking *launch* to begin the process. Once the job be completed, the results may be downloaded from the Job History page.

The image A.5 captures the Job History page, and the image A.6 shows a capture detailing job #56 from the Job Details page.

A.2 Long-term Operation

To send MapReduce workflows into qosh, the operating procedure described in section A.1.4 should be followed. Changing job options, cluster size, and main class may be accomplished by filling the form fields according to the user's needs.

By default, qosh is configured to run in debug mode. This behavior may be easily modified by altering the flag `DEBUG` within `settings.py`. Furthermore, it should be noted that qosh will operate much faster when deployed to a production-level web server.

OpenStack Compute will keep a log in `/var/log/nova` that could be read should any issue appear. By default, workflow results will be stored to `$HOME/Public` which may be changed in `settings.py`, variable `MEDIA_ROOT`.

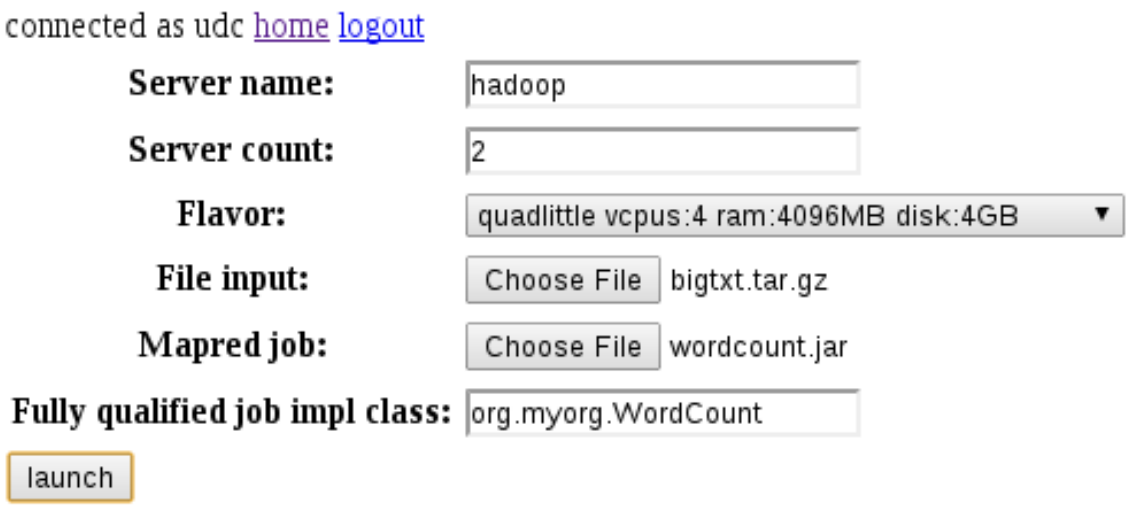

Figure A.4: MapReduce job definition

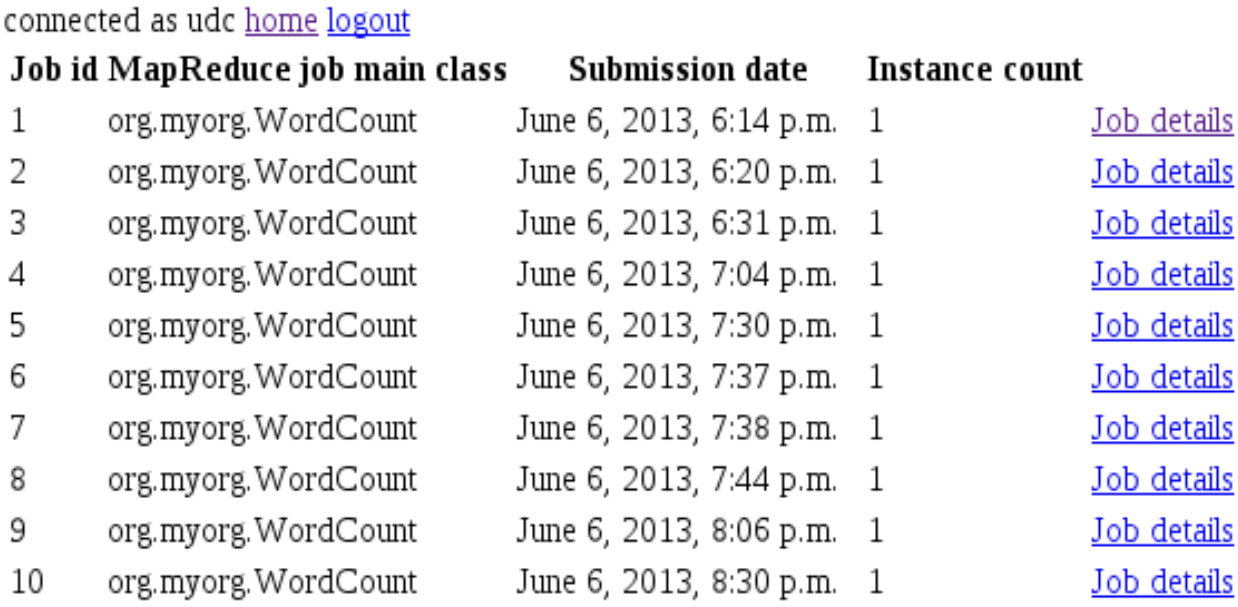

Figure A.5: MapReduce Job History

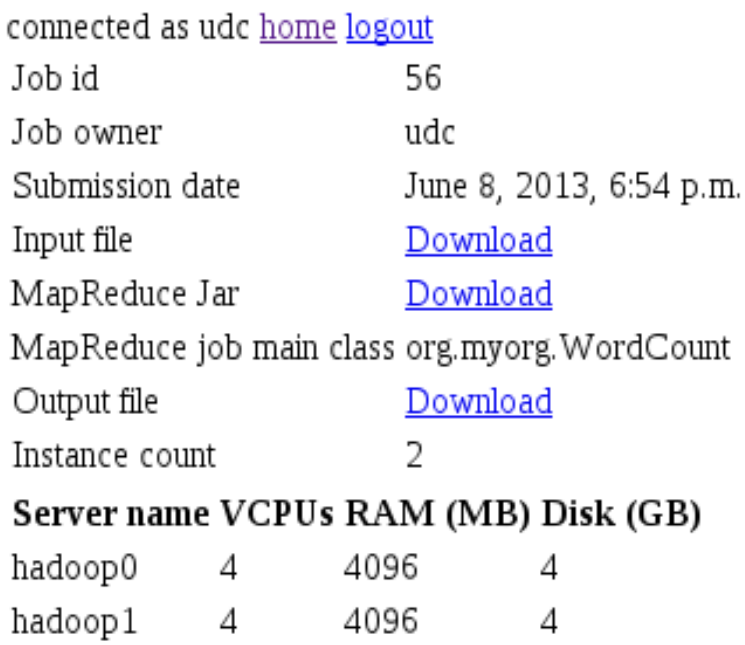

Figure A.6: Job #56 details

Appendix B

Glossary of Terms

ACPI `Advanced Configuration and Power Interface`. A power management specification that makes hardware status information available to the operating system.

AMD-V `Advanced Micro Devices Virtualization`. Set of CPU extensions implemented by Advanced Micro Devices, Inc. to support Full Hardware Virtualization.

API `Application Programming Interface`. Protocol definition interface between two or more communicating entities. It specifies an abstract format by which collaborating entities can interchange information.

APIC `Advanced Programmable Interrupt Controller`. Family of interrupt controllers used to combine different interruption signal sources into one or more CPU lines, allowing for the assignment of priority levels to each kind of interruption.

AWS `Amazon Web Services`. Set of web services offered by Amazon Inc. that let a user exploit, remotely and on-demand, Amazon's computational infrastructure.

CLI `Command Line Interface`. Refers to every application whose main interface is the terminal or command line.

CUDA `Compute Unified Device Architecture`. Parallel computing platform and programming model devised at NVIDIA and implemented in its GPUs.

DFS `Distributed File System`. Any kind of file system stored over a network of computing devices and concurrently accessible by multiple users in different points.

DHCP `Dynamic Host Configuration Protocol`. Network protocol used to automatically and dynamically configure network access parameters.

DNS `Domain Name Server`. Name service that maps certain device information from one domain to another. It is typically used to locate resources in a network by translating their names to their IP addresses.

Amazon EC2 `Elastic Compute Cloud`. Amazon, Inc.'s web service allowing for on-demand provisioning of computational infrastructure.

EPT `Extended Page Tables`. Intel's hardware virtualization system permitting a guest operating system to directly modify its own memory table pages, effectively requiring no VMM override to handle page misses.

Hard coding In Software Engineering, *Hard coding* refers to the bad practice of including an input or configuration value within the source code that controls the application logic.

Hash Function An algorithm that associates a larger variable-length data space with a smaller fixed-length data space.

HTTP `HyperText Transfer Protocol`. Application protocol for distributed, collaborative, hypermedia information systems. HTTP is the foundation of data communication for the World Wide Web.

HVM `Hardware Virtual Machine`. Virtualization approach by which a modified CPU is able to create a virtual domain for applications to exploit infrastructure resources with reduced penalty on performance. Also called *Native Virtualization*.

iptables Administrative tool and associated Linux service that allows configuring the routing tables provided by the kernel firewall and the chains and rules it stores.

IT `Information Technology`. It comprises any concept related to information and the technology to handle it, like networks, hardware, software, the Internet, as well as the people working with these technologies.

JRE `Java Runtime Environment`. Environment that need be installed in any computer requiring the execution of Java applications.

KVM `Kernel-based Virtual Machine`. Full Virtualization solution for the Linux kernel that turns it into a hypervisor. KVM requires a processor with hardware virtualization extensions.

LVM `Logical Volume Manager`. Tool set that allows a user to define an storage abstraction layer on top of conventional persistence that is more flexible than conventional partitioning schemes.

MMU `Memory Management Unit`. Hardware device, on-die in modern CPU architectures, that handles access to main memory.

NAS `Network Attached Storage`. Storage device that is attached to a preexisting network to increase the shared storage space.

NAT `Network Address Translation`. Process by which some networking devices rewrite the source and/or destination fields in telecommunications within a network segment. It is used as a means to limit the number of issued network addresses and to conceal the identity of devices behind the NAT.

NFS `Network File System`. A file system developed by Sun Microsystems, Inc. implementing client/server model that allows users to access files across a network and treat them as if they resided in a local file directory.

NTP `Network Time Protocol`. It is a networking protocol for clock synchronization between computer systems over packet-switched, variable-latency data networks.

OCCI `Open Cloud Computing Interface`. A set of specifications delivered through the Open Grid Forum, for cloud computing service providers to offer their services defining a unified interface.

QCOW2 `QEMU Copy On Write 2`. Optimization strategy by which a storage device will delay the allocation of storage space until it is actually needed.

QEMU `Quick EMUlator`. Free and open source hosted hypervisor that performs hardware virtualization.

RAID `Redudant Array of Independent Disks`. It is a data storage virtualization technology that combines multiple disk drive components into a logical unit for the purposes of data redundancy or performance improvement.

REST `Representational State Tranfer`. Client/server architectural style that defines a simple, stateless, cacheable, interface-uniform protocol used to handle resource representation in distributed systems.

RPC `Remote Procedure Call`. An inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction.

RPM `RPM Package Manager`. A package management system for the Red Hat Enterprise Linux and derivatives.

rsync Is a file synchronization and file transfer program for Unix-like systems that minimizes network data transfer by using a form of delta encoding called the rsync algorithm. rsync can compress the data transferred further using zlib compression, and SSH or stunnel can be used to encrypt the transfer.

Amazon S3 `Simple Storage Service`. Scalable, safe, fast and inexpensive storage service consumed through the Internet and provided by Amazon, Inc.

SAN `Storage Area Network`. It is a dedicated network that provides access to consolidated, block level data storage.

Sandbox In the context of software development and revision control, it is a testing environment that isolates untested code changes and outright experimentation from the production environment or repository respectively.

SELinux `Security-Enhanced Linux`. A Linux kernel security module that provides a mechanism for supporting access control security policies, including United States Department of Defense-style mandatory access controls (*MAC*).

SPOF `Single Point Of Failure`. In Systems Engineering it is the point of the architecture that if it failed it would prevent the whole system from functioning.

Amazon SQS `Simple Queue Service`. Queue service provided by Amazon, Inc. to handle asynchronous message passing between EC2 instances.

SSH `Secure SHell`. A cryptographic network protocol for secure data communication, remote command-line login, remote command execution, and other secure network services between two networked computers.

sudoers The file in a *nix system that specifies which users can execute commands as if they were the root user.

VMM `Virtual Machine Manager`. A management solution for the virtualized data center. It can be used to configure the virtualization host, networking, and storage resources, in order to create and deploy virtual machines and services to private clouds.

VT-x Intel, Inc.'s CPU extension set allowing for full hardware virtualization.

yum `Yellowdog Updater Modified`. An open source command-line package-management utility for Linux operating systems using the RPM Package Manager.

Bibliography

[1] Google Apps: Energy Efficiency in the Cloud. `http://static.googleusercontent.com/external_content/untrusted_dlcp/www.google.com/en/us/green/pdf/google-apps.pdf`, 2012. Accessed: June 2013.

[2] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified Data Processing on Large Clusters. *Common. ACM*, 51(1):107–113, 2008.

[3] Pierre Riteau, Ancuta Iordache, and Christine Morin. Resilin: Elastic MapReduce for Private and Community Clouds. Research Report RR-7767, INRIA, October 2011.

[4] Huan Liu and Dan Orban. Cloud MapReduce: A MapReduce Implementation on Top of a Cloud Operating System. In *Proceedings of the 2011 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing*, CCGRID '11, pages 464–474, Washington, DC, USA, 2011. IEEE Computer Society.

[5] Steve Loughran, Jose Maria Alcaraz Calero, Andrew Farrell, Johannes Kirschnick, and Julio Guijarro. Dynamic Cloud Deployment of a MapReduce Architecture. *IEEE Internet Computing*, 16(6):40–50, November 2012.

[6] Andy Edmonds, Thijs Metsch, Alexander Papaspyrou, and Alexis Richardson. Toward an Open Cloud Standard. *IEEE Internet Computing*, 16(4):15–25, July 2012.

[7] Rich Uhlig, Gil Neiger, Dion Rodgers, Amy L. Santoni, Fernando C. M. Martins, Andrew V. Anderson, Steven M. Bennett, Alain Kagi, Felix H. Leung, and Larry Smith. Intel Virtualization Technology. *Computer*, 38(5):48–56, May 2005.

[8] Tom White. *Hadoop, the Definitive Guide*. O' Reilly and Yahoo! Press, 2012.

[9] Nikita Ivanov. GridGain and Hadoop: Differences and Sinergies. `http://www.gridgain.com/blog/gridgain-hadoop-differences-synergies/`, 2012. Accessed: June 2013.

[10] Jaliya Ekanayake, Hui Li, Bingjing Zhang, Thilina Gunarathne, Seung hee Bae, Judy Qiu, and Geoffrey Fox. Twister: A Runtime for Iterative MapReduce. In *The First International Workshop on MapReduce and its Applications*, 2010.

[11] Aaron McKenna, Matthew Hanna, Eric Banks, Andrey Sivachenko, Kristian Cibulskis, Andrew Kernytsky, Kiran Garimella, David Altshuler, Stacey Gabriel, Mark Daly, and Mark A. DePristo. The Genome Analysis Toolkit: A MapReduce Framework for Analyzing Next-generation DNA Sequencing Data. 2010.

[12] Quizmt project web page. `http://qizmt.myspace.com/`. Accessed: June 2013.

[13] Adam Dou, Vana Kalogeraki, Dimitrios Gunopulos, Taneli Mielikainen, and Ville H. Tuulos. Misco: a Mapreduce Framework for Mobile Systems. In *Proceedings of the 3rd International Conference on Pervasive Technologies Related to Assistive Environments*, PETRA '10, pages 32:1–32:8, New York, NY, USA, 2010. ACM.

[14] Peregrine project web page. `http://peregrine_mapreduce.bitbucket.org/`. Accessed: June 2013.

[15] Bingsheng He, Wenbin Fang, Qiong Luo, Naga K. Govindaraju, and Tuyong Wang. Mars: A MapReduce Framework on Graphics Processors. In *Proceedings of the 17th International Conference on Parallel Architectures and Compilation Techniques*, PACT '08, pages 260–269, New York, NY, USA, 2008. ACM.

[16] *Eucalyptus — 3.1.1 Installation Guide*, 2012.

[17] *CloudStack Basic Installation Guide for CloudStack Version 3.0.0 — 3.0.2*, 2012.

[18] Citrix Unveils Next Phase of Cloudstack Strategy. `http://www.citrix.com/news/announcements/apr-2012/citrix-unveils-next-phase-of-cloudstack-strategy.html`, 2012. Accessed: June 2013.

[19] *Apache CloudStack 4.0.0 — Incubating CloudStack Installation Guide*, 2012.

[20] *CloudStack Advanced Installation Guide for CloudStack Version 3.0.0 — 3.0.2*, 2012.

[21] *Citrix XenServer 6.0 Installation Guide*, 2012.

[22] How to Use CloudStack without Hardware Virtualization. `http://support.citrix.com/article/CTX132015`, 2012. Accessed: June 2013.

[23] OpenNebula 3.8.1 QuickStart. `http://wiki.centos.org/Cloud/OpenNebula/QuickStart`, 2012. Accessed: June 2013.

[24] Getting Started with Openstack on Fedora 17. `http://fedoraproject.org/wiki/Getting_started_with_OpenStack_on_Fedora_17`, 2012. Accessed: June 2013.

[25] *OpenStack Install and Deploy Manual — Red Hat — Folsom*, 2012.

[26] *OpenStack Network (Quantum) Administration Guide — Folsom*, 2012.

[27] *OpenStack Object Storage Administration Manual — Folsom*, 2012.

[28] Sanjay Ghemawat, Howard Gobioff, and Shun-Tak Leung. The Google File System. In *Proceedings of the Nineteenth ACM Symposium on Operating Systems Principles*, SOSP '03, pages 29–43, New York, NY, USA, 2003. ACM.

[29] *Amazon Elastic Compute Cloud — User Guide — API Version 2012-12-01*, 2013.

[30] QEMU Internals. `http://qemu.weilnetz.de/qemu-tech.html`, 2012. Accessed: June 2013.

[31] Jaromír Hradílek, Douglas Silas, Martin Prpič, Stephen Wadeley, Eliška Slobodová, Tomáš Čapek, Petr Kovář, John Ha, David O'Brien, Michael Hideo, and Don Domingo. *Fedora 17 System Administrators' Guide. Deployment, Configuration and Administration of Fedora 17.* Red Hat Inc., 2012.

[32] Christopher Curran and Jan Mark Holzer. *Red Hat Enterprise Linux 5.2 – Virtualization Guide.* Red Hat Inc., 2008.

[33] Michael Hideo Smith. *Red Hat Enterprise Linux 5.2 — Deployment Guide.* Red Hat Inc., 2008.

[34] Johan De Gelas. Hardware Virtualization: the Nuts and Bolts. `http://www.anandtech.com/show/2480`, 2012. Accessed: June 2013.

[35] *OpenStack Install and Deploy Manual — Red Hat — Essex*, 2012.

[36] *OpenStack Compute Administration Manual — Essex*, 2012.

[37] *OpenStack Compute Administration Manual — Folsom*, 2012.

[38] Computer Desktop Encyclopedia. `http://www.computerlanguage.com`. Accessed: November 2014.

[39] Jacek Artymiak. *Programming OpenStack Compute API.* Rackspace US Inc., 2012.

[40] *OpenStack API Reference*, 2013.

[41] OpenNebula OCCI Specification 3.8. http://opennebula.org/documentation:archives:rel3.8:occidd, 2012. Accessed: June 2013.

[42] DEISA. Deisa glossary. http://www.deisa.eu/references. Accessed: June 2013.

[43] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf, 2011. Accessed: June 2013.

[44] *CloudStack API Documentation (v3.0)*, 2012.

[45] Simon Kelley. dnsmasq — A Lightweight DHCP and Caching DNS Server — Man-Page. http://www.thekelleys.org.uk/dnsmasq/docs/dnsmasq-man.html, 2012. Accessed: June 2013.

[46] HDFS Users' Guide. http://hadoop.apache.org/docs/r1.0.4/single_node_setup.html, 2010. Accessed: June 2013.

[47] Single Node Setup. http://hadoop.apache.org/docs/r1.0.4/single_node_setup.html, 2008. Accessed: June 2013.

[48] MapReduce Tutorial. http://hadoop.apache.org/docs/r1.0.4/mapred_tutorial.html, 2008. Accessed: June 2013.

[49] Cluster setup. http://hadoop.apache.org/docs/r1.0.4/hdfs_user_guide.html, 2008. Accessed: June 2013.

[50] Dhruba Borthakur. HDFS Architecture Guide. http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html, 2012. Accessed: June 2013.

[51] Jimmy Lin and Chris Dyer. *Data–Intensive Text Processing with MapReduce*. 2010.

[52] Sriram Rao, Raghu Ramakrishnan, Adam Silberstein, Mike Ovsiannikov, and Damian Reeves. Sailfish: A Framework for Large Scale Data Processing. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, pages 4:1–4:14, New York, NY, USA, 2012. ACM.

[53] Ahmed Metwally and Christos Faloutsos. V-SMART-join: A Scalable MapReduce Framework for All-pair Similarity Joins of Multisets and Vectors. *Proc. VLDB Endow.*, 5(8):704–715, April 2012.

[54] Amr Awadallah. Apache Hadoop in the Enterprise — Keynote. Cloudera Inc., 2011.

[55] Mendel Cooper. Advanced Bash-Scripting Guide. An In-depth Exploration of the Art of Shell Scripting. http://tldp.org/LDP/abs/html/, 2012. Accessed: June 2013.

[56] Bruce Barnett. Sed — An Introduction and Tutorial. http://www.grymoire.com/Unix/Sed.html, 2012. Accessed: June 2013.

[57] MapReduce Wikipedia entry. http://en.wikipedia.org/wiki/MapReduce, 2012. Accessed: June 2013.