UNIVERSIDADE DA CORUÑA

SCHOOL OF INFORMATICS

Department of Electronics and Systems

Assessment, Design and Implementation of a Private Cloud for MapReduce Applications

Author: Marcos Salgueiro Balsa

Tutoring: Patricia González Gómez
Tomás Fernández Pena
José Carlos Cabaleiro Domínguez

Date: A Coruña, June 2013

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

Great spirits have always encountered violent opposition from mediocre minds.
Albert Einsten

The supreme art of war is to subdue the enemy without fighting.
Sun Tzu, *The Art of War*

[...] 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 technology's unending evolution has promoted changes in its ways and means. Today, *tablets* and *smartphones*, quantitatively inferiors 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 like to have all of their data synchronized and at hand, all the time.

But that is only a part in the equation. At the other side of every service request there lays a server that must deal with an ever increasingly troubling traffic volume, while it maintains response delivery at 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 implementation effort with energetic efficiency is a transcendental requisite and not simply a good practice, we would have a perfect environment for the proliferation of new distributed paradigms as the *Cloud*. The Cloud is not an intrinsically new idea but an old concept abstraction: *virtualization*. The clouds' cornerstone is flexibility.

Another technology that is constantly making it to the 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; with makes them an extraordinary computational tandem. This project puts forth a solution that allows for drawing on computational resources available exploiting both technologies together. 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 *OSS*, favoring its usage as starting point for larger installations.

Keywords

Distributed Computing, Virtualization, Cloud Computing, MapReduce, OpenStack, Hadoop.

Contents

List of Figures

Chapter 1

Abstract

Over the last years there has been a continuous increase in the quantity 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 — relational database systems mostly — 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 general underutilization — only 10% of Amazon's global computational resources were in use — that pushed the search for alternative means to export the 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 client's

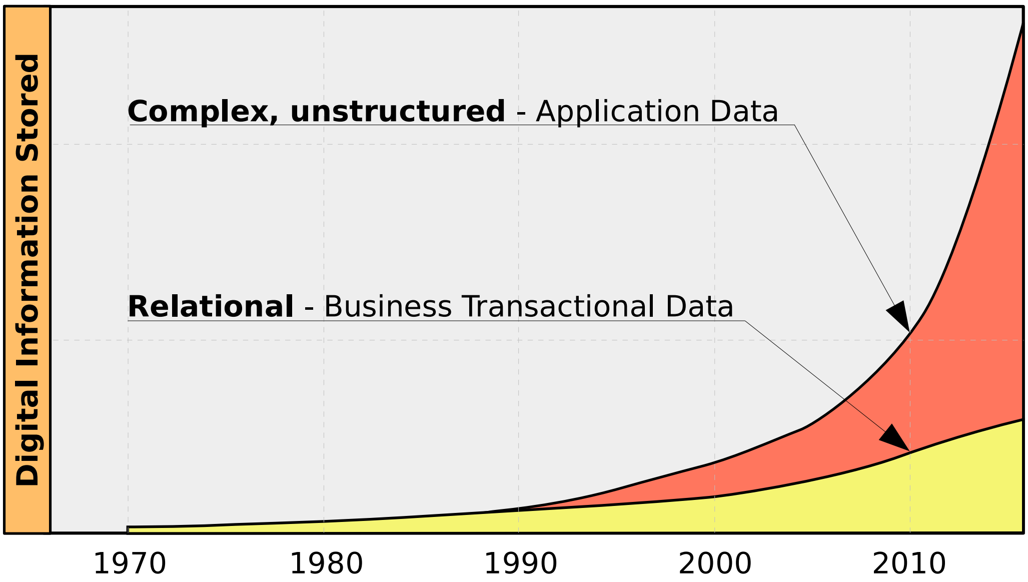

Figure 1.1: Demand in exponential growth. Source: *Cloudera Inc.*

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 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, ecommerce, image processing applications or searching for new forms of energy.

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 processings

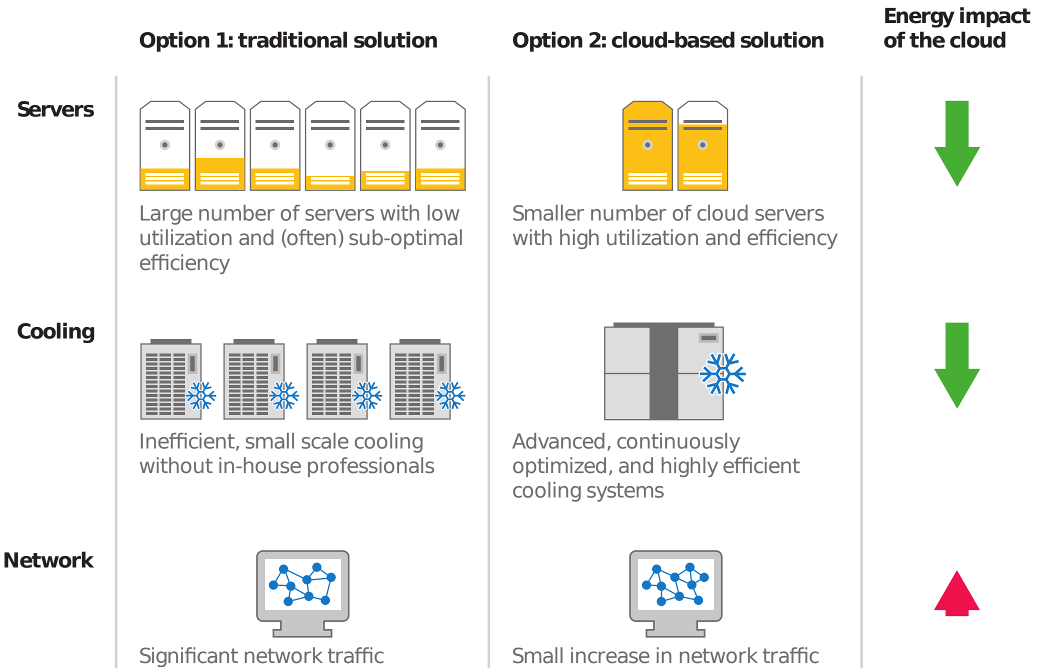
Option 1: traditional solution
Option 2: cloud-based solution
Energy impact of the cloud
Servers
Large number of servers with low utilization and (often) sub-optimal efficiency
Smaller number of cloud servers with high utilization and efficiency
Cooling
Inefficient, small scale cooling without in-house professionals
Advanced, continuously optimized, and highly efficient cooling systems
Network
Significant network traffic
Small increase in network traffic

Figure 1.2: Energy savings. Source: [1]

(Figure 1.2).

1.1 Goals

The main goal with this project 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 on 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 to this human-cloud-mapreduce interface. 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.

1.2 Arrangement of the Document

The contents within this document are distributed as stated next. This first chapter introduces development guidelines in the abstract. Chapter 2 puts the reader closer to the fundamental Cloud Computing concepts — like its general architecture or virtualization —, along with the ones 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. Analogically, chapter 5 unveils Hadoop's peculiarities as a MapReduce implementation.

The subsequent chapters center on detailing the project from diverse vantage points. Chapter **??** contains a series of design decisions and their accompanying UML diagrams. Chapter **??** gathers an analysis on performance in a real testing cluster. Chapter cap:conclusiones analyzes related papers 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 **??** guides the reader throughout a quick single node installation. Annex **??** 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 successive elaborations to the project will lay on top of them.

2.1 Cloud Computing

In essence, Cloud Computing, or Cloud for short, is a distributed computing model that attempts to ease the consumption on-demand of that distributed infrastructure, by exporting it as virtual computational resources, platforms or services. However it may seem, the Cloud is no new technology but it introduces a new manner to exploit idle computing capacity. What it intends is to make orchestration of enormous data centers more flexible, so as to allow a user to 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

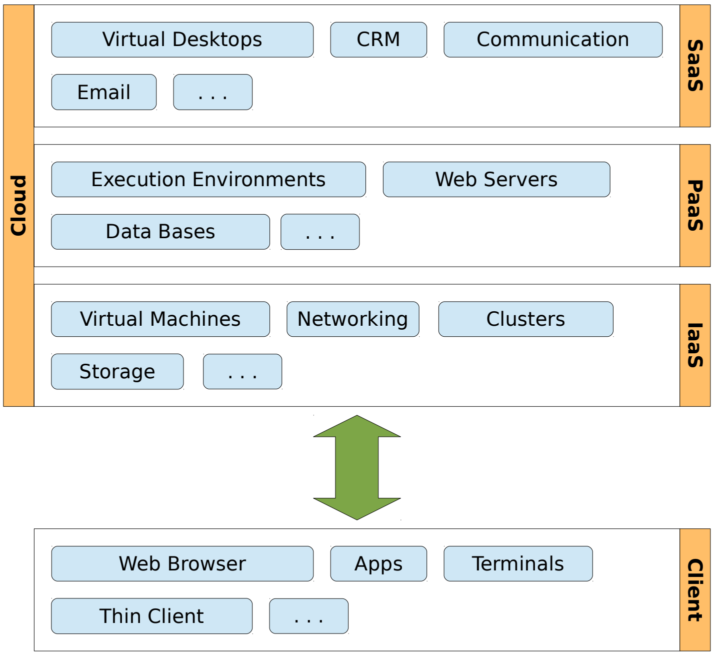

Figure 2.1: Layers in a cloud in production

be needed, bearing in mind that the degree of elaboration grows from IaaS services to SaaS ones. In effect, PaaS and SaaS layers are lied supported by those immediately under — software 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 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, bringing up the software layers that give form PaaS and SaaS paradigms respectively.

No less important cuestions like access control, execution permissions, quota or persistent or safe storage will also be present in all of the frameworks.

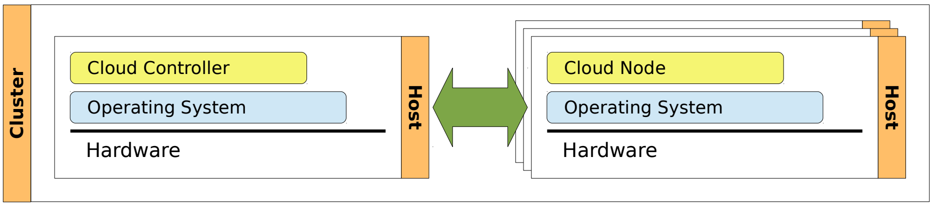

Figure 2.2: Cloud Controller and Cloud Node

2.1.1 Architecture

Figure 2.1 showed possible layers that could be found in a cloud deployment. Depending on the layers that are implemented, the particular framework and the role played by the cluster node, different particular modules will appear to make possible the consumption of 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 dictating how those subsystems should be in terms of size and responsibility, and thus, 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 lies close to *Master-Slave* architectures' approach. In those, in the abstract, 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.

Within this general role distribution in a cluster, host computers or cluster nodes — labeled as Cloud Controllers or Cloud Nodes — cooperate in a synchronized fashion through *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 set sharded (distributed) between the members of the set.

Although there is no practical restriction to configuring both Cloud Controller and Cloud Node within a single computer in a cluster, this approach should be limited to development environments due to the considerable impact in performance that it would carry.

Cloud Controller

The fundamental task for a Controller is to maintain all of the cloud's constituent modules working together by coordinating their cooperation. As an example, it is a Controller's duty to:

- Authentication and authorization control.
- Available infrastructure resources recount.
- Quota management.
- Usage balance.
- User and project inventory.
- API exposure for service consumption.
- Real time cloud monitoring.

Being an essential part of a cloud as it is, the Controller node (not to be mistaken for the Cloud 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 clouds through a Web Service API — mostly *RESTful* APIs (*REpresentational State Transfer*). Those APIs vary slightly from company to vendor as usual, which forces clients to

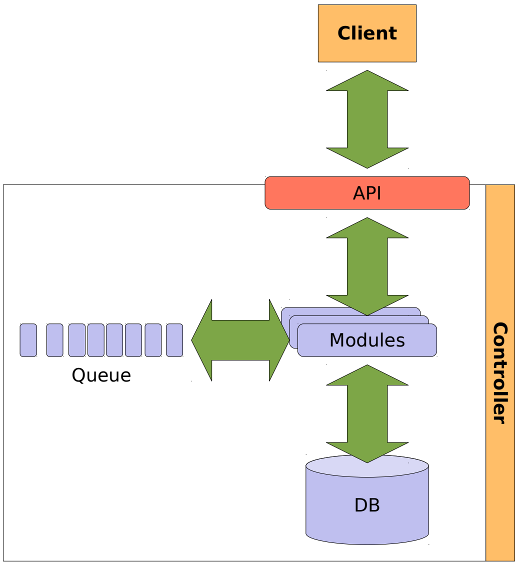

Figure 2.3: Cloud Controller in detail

be partially coupled to clouds. 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 conforming modules support its functional requirements. Each one of them will have a well-defined responsibility, and so appear networking modules, access and security control modules, storage modules, etc. Many of them existed before the advent of Cloud Computing but they worked only locally. Inter-module communication is handled by means of an asynchronous message queue that guarantees an equally efficient broadcasting system outside of the Cloud Controller, i.e. the rest of the cluster nodes participating in the cloud.

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 whose profiling must be properly tailored.

Hardware requirements on the cluster nodes vary from each particular framework implementation and the *QoS* expected, but, in the abstract, they normally need something around 10 GB of RAM, quad core CPU, Gigabit Ethernet and one TB of storage.

Cloud Node

If the Cloud Controller is entrusted the cloud's correct functioning acting like a glue for its parts, the actual task processing is performed in the Cloud Nodes; that is, the VCPU, VRAM, VHDD are going to be mapped 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 the 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. Further, this homogeneous space could be provisioned, as discussed above, on demand. It is the Cloud Controller's responsibility 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 the former 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 following the AWS nomenclature — by creating the virtual architecture needed and a *virtual execution domain* managed with the help of the operating system kernel. To generate this architecture there fundamentally exist three techniques: *Emulation*, *Paravirtualization* and *Hardware Virtualization* or *Full Virtualization*. Different hypervisors will support them in a different degree, but most will cover only 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. Besides, the kernel in the guest operating system — the kernel in the virtual machines's — operating system needs no alteration whatsoever, and the cluster node's 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 imposes a special treatment to be given to its 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 the convenient hardware extensions are present (SVM and *VT-x* respectively [7]). Just as what happened with emulation, an unaltered host's 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 installed 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.

Paravirtualization

Paravirtualization uses a different approach. To begin with, it is indispensable that the guest's kernel be modified to make it capable of interacting with a paravirtualized environment. When the guest runs, the hypervisor will separate those 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 processes. Subsequently, the hypervisor will manage an on-contract execution between host and guest allowing the latter to run kernel mode sections as if pertaining to the real execution domain — as if they were processes running in the host, not in the guest — with almost no performance slowdown. Paravirtualization, in turn, does not require an special hardware extension be present.

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 frames. This fact has raised a condition of lacking outwards interoperability, maturing non-standard APIs; though today those divergences are fading away. These frameworks and APIs are product of the efforts to improve and ease controlling the underlying particular clusters on which they germinated. Thus, it is no surprising their advances had originated parallelly with the infrastructure they drove, leaving compatibility in the background.

Slowly but steadily these managing systems became larger in reach and responsibility boosted by an increasingly interest in the sector. In the end, it happened that software and systems engineering made them more abstract, so they finally overlapped functionally. AWS appearance finished forging the latent standardization need, and thus, as of today, most frameworks offer APIs closer and closer to Amazon's — nowadays the de-facto standard —

and OCCI's [6].

2.2 MapReduce Paradigm

The origin of the paradigm centers around a paper publication of 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 execution 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].

2.2.1 Programming Model

The MapReduce programming model requires the developer express his problem 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 give rise to *Map* and *Reduce* functions. Both Map and Reduce have to be supplied by the developer, which may force a deviation in breaking the original problem down. As counterpart, the MapReduce model will deal with parallelizing the computation, distributing input data across the cluster, handling exceptions that could raise and recovering output results; everything transparent to the programmer.

$$\boxed{\begin{array}{l}\mathbf{map} : (\alpha \rightarrow \beta) \ \rightarrow \ \alpha \, list \ \rightarrow \ \beta \, list \\ \mathbf{map} \ (\mathbf{pow}\, 2) \ [1,2,3] \ \Rightarrow \ [1,4,9]\end{array}}$$

Figure 2.4: Map function example (functional version)

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

Figure 2.5: Map function signature (MapReduce version)

Función Map

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, map function receives a tuple as input and produces another tuple *(key, value)* as intermediate output. It is the MapReduce library who is responsible for feeding the map function by mutating the data contained in input files into *(key, value)* pairs. Then, 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 figure 2.5.

Reduce function

The typical functional fold expects any function G, a list L, or generally any type of recursive data structure, and any initial element I, subtype of L's elements. Fold returns the value in I resulting from building up the intermediate values generated after applying G to each element in L. Figure 2.6 presents fold signature as well as an example.

$$\begin{aligned}&\mathbf{fold} : (\alpha \to \beta \to \alpha) \;\to\; \alpha \;\to\; \beta\, list \;\to\; \alpha\\ &\mathbf{fold}\ (+)\ 0\ [1,2,3] \;\Rightarrow\; 6\end{aligned}$$

Figure 2.6: Fold function example

$$\begin{aligned}&\mathbf{reduce} : \ (k2, v2\, list) \;\to\; v2\, list\\ &\mathbf{k} : clave\\ &\mathbf{v} : valor\\ &(\mathbf{kn}, \mathbf{vn}) : par\ \ (clave, valor)\ \ en\, un\, dominio\, n\end{aligned}$$

Figure 2.7: Reduce function signature

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 from subsequent maps. Reduce 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 within the same grouping 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 with 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 each word found. Thus, an explosion of intermediate pairs will be generated as

```
Map (String key, String value):
// key: documento 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]

output of map, 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 on each invocation the pair *(<word>, list(“1”))*. 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 — this supposing that the map phase were over before starting the reduce phase and that every word *<word>* were submitted to the same reducer in the cluster — a cluster node executing the reduce function.

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 parading 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 identity function.

- Count of URL access frequency: Like wordcount.

- 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 generates the sequence of pairs *(<word>, list(document_ id))*.

2.2.3 Processing Model

Besides defining the structure that the applications willing to leverage the 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, it can be stated that task processing on each phase is done in parallel and phases execute in sequence; yet, it is not 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.

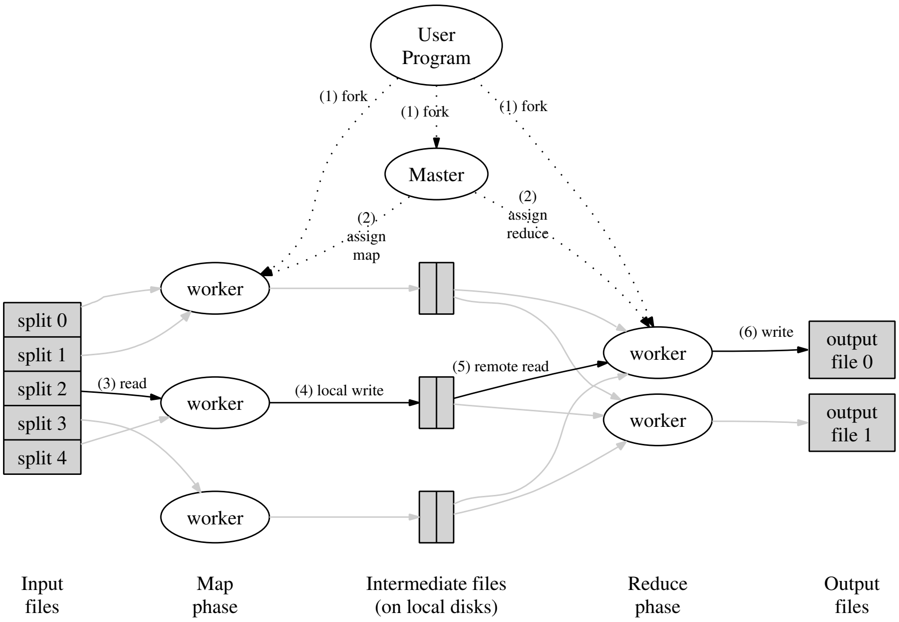

Figure 2.9: MapReduce execution diagram. Source: [2]

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

2. From this moment each program copy resides in a cluster node. A random copy is chosen among them and 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 from the master node. There will be M map tasks and R reduce tasks.

3. Workers assigned with 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 a cache.

4. Periodically, those pairs in memory are dumped to a local disk — dumped to a drive of the node that is executing the map function — and partitioned into R regions. Their path on disk is then sent back to the master, responsible for forwarding these paths to *reduce workers* or *reducer*s.

5. Now, when a reducer is notified that it should start processing, the path to the data of the reduction is send along and the reducer will fetch them 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 to the key. The output fo the reduce function for the reducer partition 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 indeterminatedly 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 within an execution flow and duly implements a series of actions against them.

Worker Failure

The least taxing of the problems. To control that every worker is up, the master node pings 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 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 also the benefit of lowering the probability of failure, precisely why in the original paper [2] it had been put forward that the entire job be canceled. Still, as there is no good design 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 a modern deployment is network bandwidth. In MapReduce executions, the information flows into the cluster from the 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 wide enough channel 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 data in the mappers' and reducers' local drives —, effectively diminishing the transport over the net.

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. And 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 few tasks are left incomplete for a particular job, *Backup Task*s are created an submitted to additional workers, making a single task be executed twice concurrently. By the time one copy of the task succeeds it will 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 — and 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 of 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 set up only once. Thus, *Twister* really shines in executing *iterative* MapRecude jobs — those jobs where maps and reduces do not happen in sequence once, but need instead a multitude of complete map-reduce cycles to succeed.

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

Qizmt [12] Written in C# and deployed in 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 throughout 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 graphic 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 storing, have been reporting back an increasing interest from the IT industry. This has brought out many pluggable extensions that enhance Hadoop's applicability.

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 one remains. A deep study will be carried out on that prevailing one.

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, is large and complex. Besides, trying to *emulate* the real hardware that will support the cloud is meager at times, e.g. if full hardware virtualization were used, the hypervisor would have to be allowed direct access to the CPU. *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

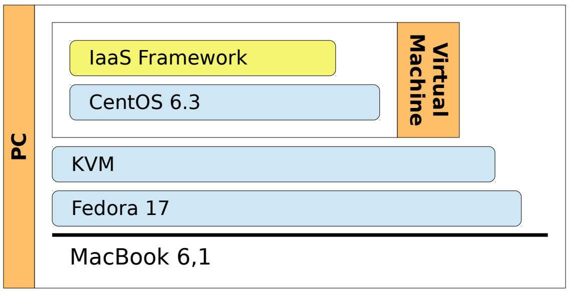

Figure 3.1: General testing space

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 born in mind.

The testing environment follows the organization shown in figure 3.1.

3.2 Evaluated Frameworks

The frameworks are:

- CloudStack
- Eucalyptus
- OpenNebula
- OpenStack

From an exclusively functional vantage point, the four of them cover clearly the requirements imposed with the project, which, in short, it would be the faculty to run and manage the lifecycle of an indeterminate number of custom VMs, tailored for MapReduce executions, through an API that would allow for the definition of a simple job control interface.

Eucalyptus and *OpenStack* take a more modular approach to the solution, unveiling smaller functional parts, while at the same time decoupling those modules. 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, the framework of choice 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 different degree — some of them partially support it, others use *adaptors* to it. OpenNebula, OpenStack and Eucalyptus have demonstrated to be carrying on coding efforts to fully support the OGF's standard: OCCI.

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 set up 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

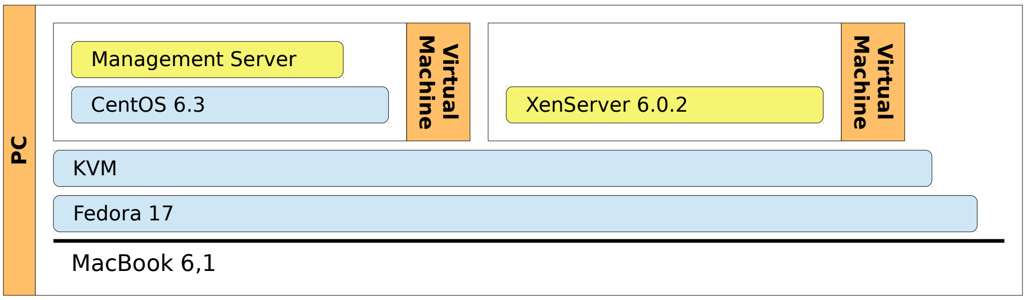

Figure 3.2: CloudStack 3.0.2 with XenServer hypervisor

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 ([20]), made us arrange the layout shown in figure 3.2.

Following the advanced and quick installation guides — [17] and [21] — the process may be summarized in the steps bellow:

- Two VMs were created to contain CloudStack *Management Server* (*MS*) and 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 contained in the quick installation guide and in the XenServer configuration manual [?] were followed to perform the configuration.

- Additionally:
  - Before specifying the execution environment, defining the cluster, primary and secondary storage, etc. a global flag had to be set to permit nodes with no virtualization extensions [19].
  - 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 image was loaded to the cloud from the 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.

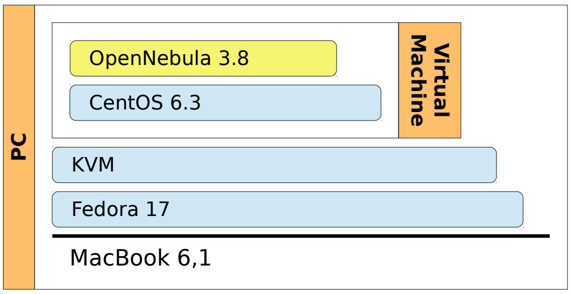

Figure 3.3: OpenNebula 3.8 with KVM

- `SELinux` was stopped.
- The configuration guide ([23]) was followed to complete the process. Afterwards, the next series of actions were carried out 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 networking interface *eth0*. The very same happened with `occi`, the REST service that exposes the cloud API.
  - Furthermore, iptables was modified for letting through traffic 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.

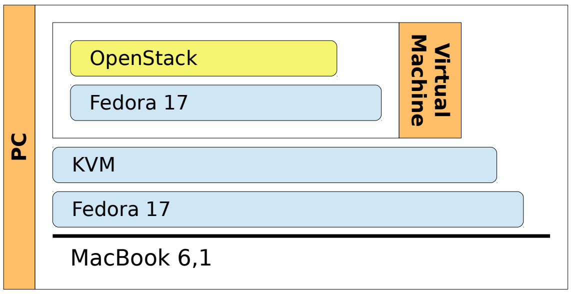

Figure 3.4: Virtual OpenStack deployment

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, even though the official *OpenStack Folsom* configuration guide [25] is written with Ubuntu in mind so many commands are not applicable to Fedora.

Both Essex and Folsom version were tested. The reason was 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 underlying state of the infrastructure, under the hood, the core module had 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 a 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 to destroy the instance requiring the invocation of console commands. This hanging problem is solved in Folsom.

An execution environment for both versions was spawned 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 explaining the findings in the comparison between the frameworks according to the methodology explained in the beginning of this section.

Installation: Without a doubt OpenNebula stands out. The installation guide is the lighter and shortest to follow with difference. 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 keep in check 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 [21] describes a first approach toward configuring Swift as secondary storage for the cloud. This fact speaks volumes about the maturity and importance of Swift, an OpenStack module.

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 of them 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 documented backing services does not correspond to the real ones; not such thing happened for Ubuntu. Nonetheless, inaccuracies are trivial to cope with and the manuals are deep enough to deploy in production.

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 the cloud over a virtual cluster 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 constituents 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 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

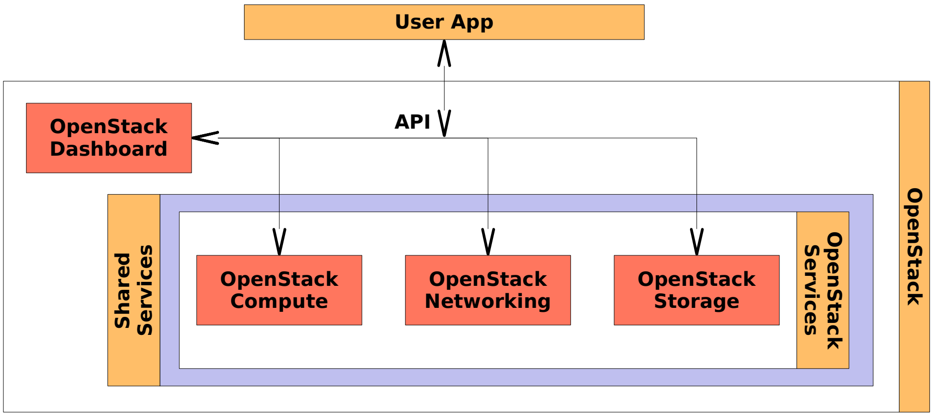

Figure 4.1: OpenStack Arquitecture

deployment is shown in figure 4.2; OpenStack's own modules are displayed in red, supporting services are shown in violet. What it is missing from the diagram, for clarity, is the asynchronous queue that mediates inter-module communication. Qpid and RabbitMQ are the two queue implementations that are officially documented, being the former the one that we used in our test deployment.

4.2 Horizon

Horizon represents the fundamental window to set up the cloud. As discussed in the previous section, Horizon does not currently — as of Folsom version — present a global view of the physical infrastructure, leaving the user in the dark in this respect. Horizon is written in Python on top of `Django`, the 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 (`noVNC`) system to view the output of the virtual graphic 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 con-

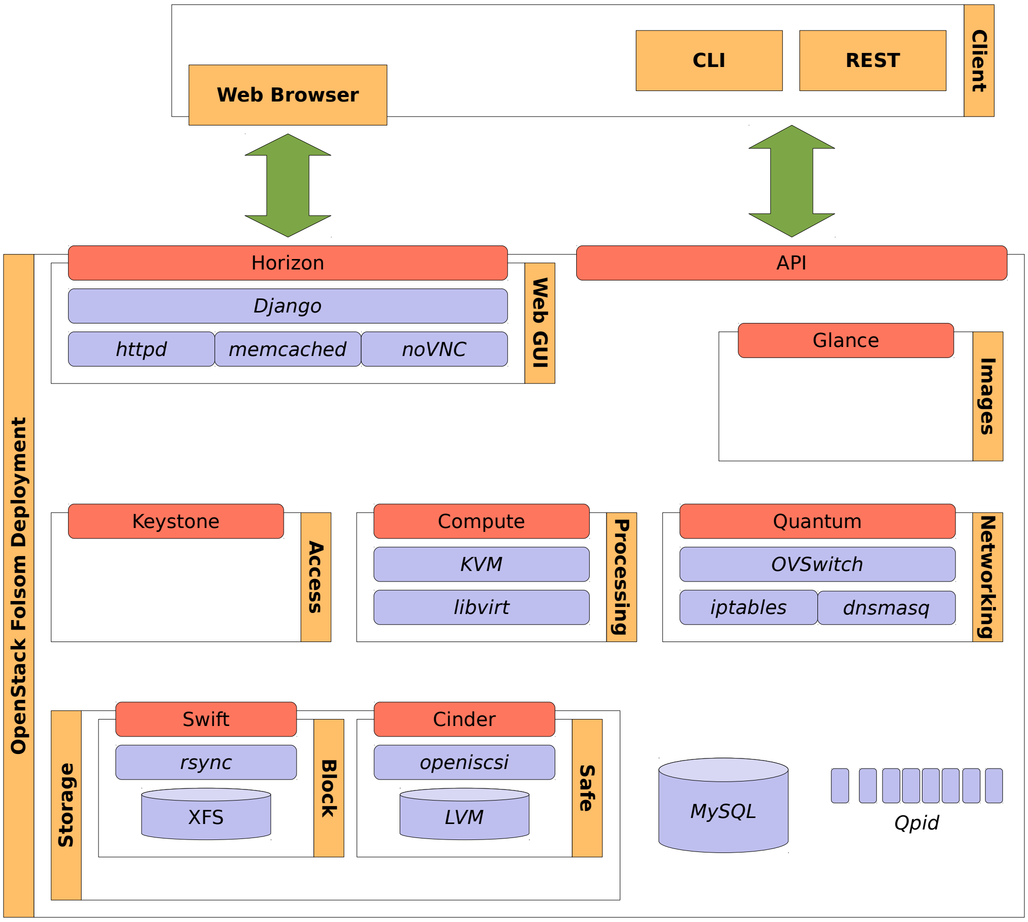

Figure 4.2: Example of an OpenStack Folsom deployment

sume those services whose role give access to. While the admin is allowed to sign up custom roles, two roles that ship 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.

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 information repository storing information needed to access the cloud installed services. It verifies, before each request, user credentials and authorizations in OpenStack services. Keystone divides this functionality in two parts: on the one hand user control, on the other service catalog.

To deal with users, Keystone assigns them 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 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 to 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 to consume a service: *Create a new instance.*

Stemming from the fact the Horizon exposes only a part of OpenStack functionality, to help dealing with security Keystone installs a CLI tool to interact with the REST service in charge of administrative operations. Issuing certain commands to Keystone through a terminal 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 metadata.

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, held together in `Compute` module. Certainly, the fact that it had been

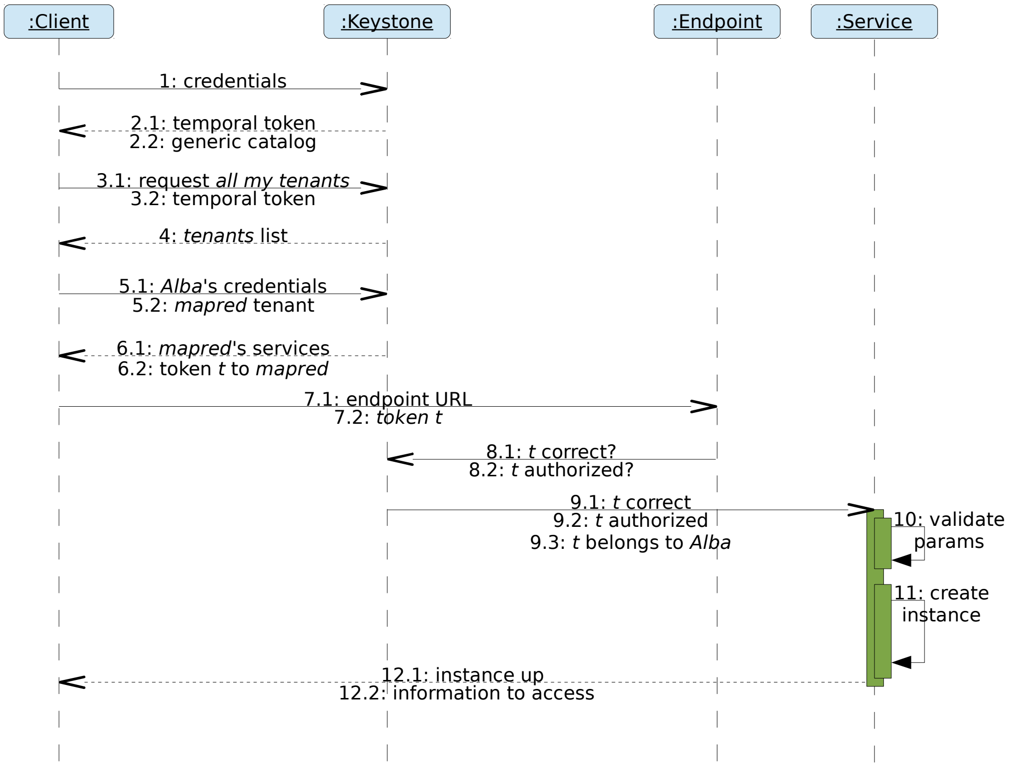

Figure 4.3: Sequence Diagram — create instance

refactored out demonstrates OpenStack's evolving model toward a more coherent less coupled functional allocation; and as it is independent, it could be configured in a dedicated node.

To bring virtual networking into existence Quantum banks on external plug-ins. Two of those plug-ins whose usage 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 the *DNS*, the *DHCP* and the *NAT*.

Figure 4.4 pictures a topology example of a virtual network. On it, *30.0.0.X* represent public IPs and *10.0.X.Y* private. This virtual network assigns a virtual router to each tenant but more 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 assigned more external interfaces — must be taken from the external network (*30.0.0.0*).

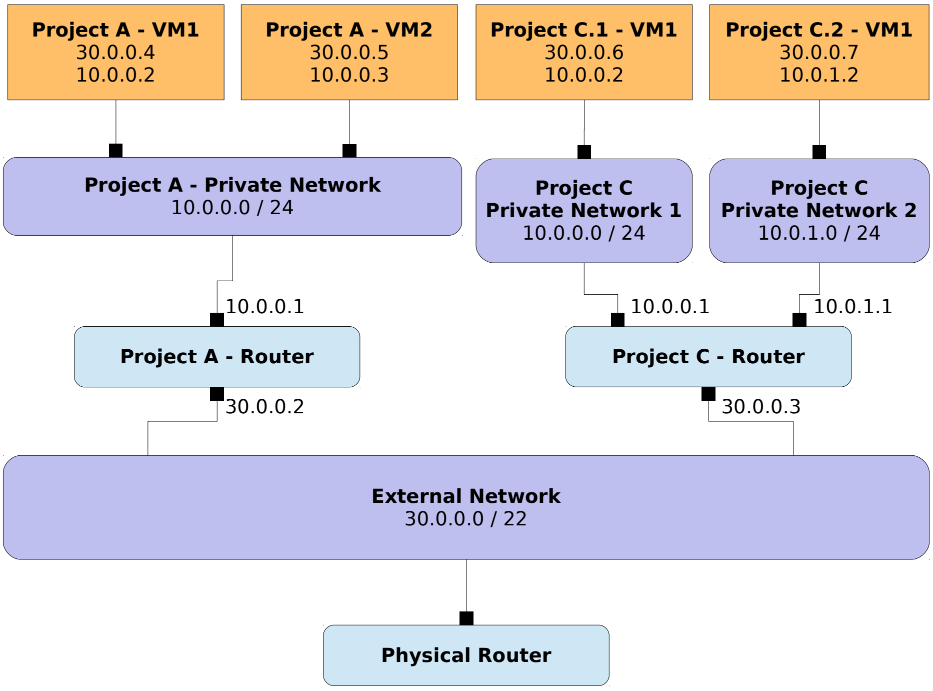

Figure 4.4: Virtual network deployment with Quantum

4.5 Compute

Compute is the central module. Its duty entails orchestrating the global workings in the cloud, delegating each particular function to the service 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, Compute does not contain a virtualization package. The approach is to delegate infrastructure provision to a hypervisor found typically, 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).

To create an instance in effect, Compute will communicate with other modules within the cloud to orchestrate the execution and, finally, it will pass the request to the most suitable cluster node's hypervisor — most suitable according to the cloud-defined rule set — that will bring up the VM. Some of the supporting services are described bellow.

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 to their requirements with ease.

4.6 Glance

Glance is OpenStack's OS 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 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 to the Cloud framework as it is the hypervisor who would have to support the particular combination image type, container type —, and they are stored as metadata linked 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 the execution of the VM. Any change on the image files is written temporarily to be discarded as soon as the VM is shut down.

Block: By making use of storage volumes managed by Cinder with *LVM* (*Logical Volume Manager*) OpenStack provides the ability to attach indeterminably-sized logical volumes to instances on-demand. This store kind guaranties 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 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.

Safe: Swift manages a safe distributed storage banking on controlled replication allowing for high availability deployments that overcome hard drive's inherent fragility. Swift draws on `rsync` to synchronize *XFS* partitions.

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 to instances, as *NFS* (*Network File System*) or a *SAN* (*Storage Area Network*) solution do; for a single volume cannot be attached to different instances at the same time. An interesting option that Cinder opens is using a logical volume to boot instances, therefore sharing the set of files contained in the image among them.

4.7.2 Swift

Just as happened with Cinder, Swift cannot be framed into traditional shared networking 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 partially

Amazon-compatible REST API. Central to Swift's implementation is replication.

Replication

Scalability, fault tolerance, high availability, safety, 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 block copies at different locations of the deployment to guarantee better performance and limit failure impact.

Within Swift, replication processes on every *Object Server* — any node in the cluster configured to support them — 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 will trigger automatic rebalancing. 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 to make a data request stay unreplied. It happens then that the execution of this operation 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 quarantined and replicas would be made anew.

Chapter 5

Hadoop

This chapter tries to expose with simplicity the defining fundamentals of Hadoop architecture. Initially abstract concepts will be introduced to give way to more particular and deep ideas that explore Hadoop implementation of the MapReduce model in two layers: the processing 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. 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 the *old* 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 pcs that greatly simplified its deployment. Nutch will inherit a large part of the concepts detailed there translated in their own distributed file system implementation (*NDFS*).

Also in 2004 appeared another publication [2] that presented MapReduce, bringing about successive efforts to port Nutch algorithms to adapt to the emerging model. In mid 2005 most of Nutch code run following MapReduce guidelines over NDFS.

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

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

5.2 General Hadoop Architecture

Hadoop composition differentiates 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 Hadoop MapReduce. It is employed to manage resources and schedule executions in distributed environments.

Hadoop DFS: The distributed file system sustaining inputs and outputs from Hadoop clusters.

Hadoop architecture corresponds to the *Master-Worker* archetype where two roles on each cluster appear: a unique Master and various Workers. These roles, and thus responsibilities, are fixed to different nodes by the cluster admin. If necessary, e.g. for maintenance, the admin may freely reset the roles to new cluster nodes, only requiring job resubmissions if the Master role were reassigned.

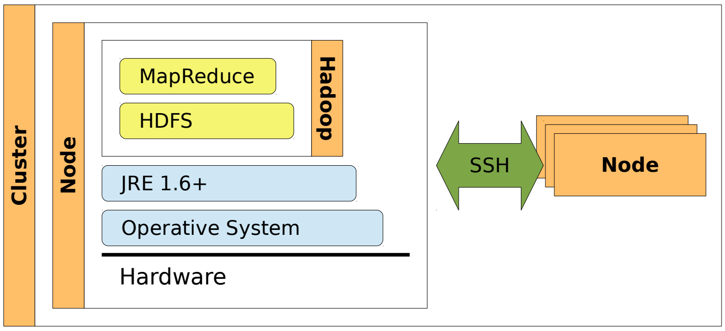

Figure 5.1: Hadoop over HDFS

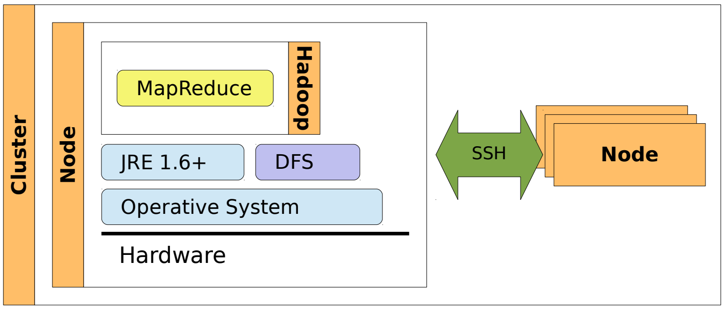

Figure 5.2: Hadoop over another DFS

This section will almost exclusively center around MapReduce and Hadoop DFS (HDFS) modules to expose the functionally covered with Hadoop. Hadoop YARN, as discussed, is a subsystem resulting from the isolation of scheduling and processing, both found together in the old — pre Hadoop 2 — MapReduce module, retaining task distribution and planning within YARN. This way, YARN is allowed to untie from Hadoop allowing for deployments where YARN orchestrates an implementation-agnostic working set. As of this writing, Hadoop YARN is still an alpha version.

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 supporting distributed file system.

From the figures it can be deduced that Hadoop runs atop a *Java Virtual Machine* (emphJVM), that MapReduce requires a DFS implementation to rely on and that inter-node communication is conveyed through *SSH* tunnels over TCP. Every module includes a web server (*Jetty*) to ease collecting and reporting status information

De las figuras se deduce que Hadoop corre sobre una máquina virtual Java, que MapReduce precisa un *DFS* (*Distributed File System*) subyacente, ya sea HDFS o alguno de los soportados, y que la comunicación internodal se realiza usando RPC sobre TCP/IP a través de un túnel *SSH* (*Secure SHell*). Todos los módulos de Hadoop contienen un microservidor web (*Jetty*), para facilitar la recolección de información de estado de cada uno.

5.3 Hadoop Distributed File System

El sistema de ficheros distribuido de Hadoop (HDFS) está diseñado para archivar gigantescas masas de datos (TeraBytes, PetaBytes, etc.), cuyo patrón primario de acceso sea *escribir-una-vez*, *leer-muchas*. No es requisito que la información albergada sea accedida exclusivamente siguiendo este patrón, pero la implementación de HDFS potencia los accesos de este tipo. El hardware subyancente no tiene ningún requisito especial y HDFS lo transforma en un repositorio de datos robusto, tolerante a fallos, fácilmente escalable, con balanceo de carga automático, reducción del ancho de banda de red consumido, etc. Sin embargo, dado que la capa física soporte no tiene ningún carácter fuera de lo común —normalmente clusters formados por nodos en red con almacenamiento local—, en HDFS confluyen algunas limitaciones operativas:

- Alta latencia de acceso al dato. La máxima de HDFS es priorizar las lecturas grandes, y así se obtienen tiempos de acceso elevados en favor de un mayor ancho de banda.

- Alta latencia de escritura de ficheros de pequeño tamaño. Derivado de la primera limitación al tener que invalidar multitud de bloques de disco, en el peor caso uno por fichero, y redistribuir las versiones actualizadas en el clúster.

- No se soportan ni múltiples procesos escritores en un solo fichero ni escrituras a disco que no sean *append* —escrituras por el final. HDFS no es un sistema de ficheros *POSIX* (*Portable Operating System Interface*), sólo implementa la parte necesaria para optimizar el procesado de datos distribuidos, siguiendo el comentado patrón de acceso.

Para organizar el almacenamiento, HDFS utiliza el concepto de bloque como en los sistemas de ficheros tradicionales. Los bloques HDFS abstraen la organización concreta de los datos en disco con una doble finalidad:

Reducir la complejidad: la escritura de un bloque comprende almacenar los datos y gestionar los metadatos asociados, como la información de localización del dato. Al utilizar el bloque como unidad organizativa, la expresión de la localización de los datos se simplifica y puede ser gestionada por otra entidad —lo que favorece la paralelización.

Aumentar la flexibilidad: nada impide que un fichero sea mayor que cualquier disco del clúster.

Para disponer los bloques en los discos locales de cada nodo del clúster, HDFS se vale de una serie de subsistemas: el *DataNode* y el *NameNode*. Además, como soporte, las últimas versiones de Hadoop permiten desplegar opcionalmente un *Backup Node* y un *Checkpoint Node*.

5.3.1 Roles de los nodos

La figura 5.3 muestra un despliegue de HDFS con la nomenclatura habitual en capas. En línea punteada aparecen representados tanto el Backup Node como el Checkpoint Node para expresar su carácter opcional.

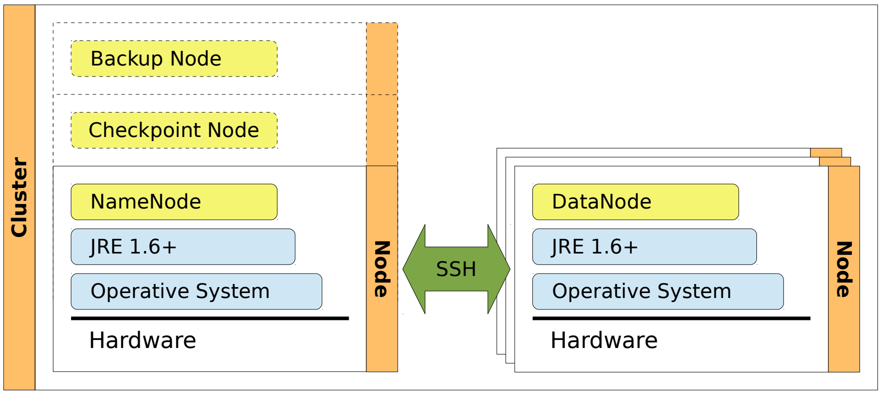

Figure 5.3: Despliegue típico de HDFS

DataNode

Son aquellos nodos del clúster encargados de almacenar en sus discos locales los bloques que contienen los datos de los ficheros del HDFS. Cada vez que escriben un bloque, porque se esté añadiendo o actualizando, se comunican con su NameNode asociado para que éste lleve la cuenta de los cambios según se van produciendo.

NameNode

El NameNode es el encargado de manejar el espacio de nombres del clúster, gestionando el árbol del sistema de ficheros y los metadatos que hacen posible recopilar la información almacenada. Es una parte tan indispensable para el acceso a los datos que, si dejase de funcionar permanentemente el nodo que presta el servicio, toda la información se perdería, ya que sería inviable saber la relación entre bloques y ficheros. Por ello, el NameNode se refuerza con nodos de puntos de restauración (Checkpoint Nodes) o con un nodo de copia de seguridad (Backup Node).

La información sobre el sistema de ficheros, los metadatos, se guarda en el NameNode de forma persistente, tanto en memoria como en disco, en dos ficheros: uno contiene la imagen del espacio de nombres (`fsimage`) y el otro el *log* de modificaciones de la imagen (`edits`). Cuando el NameNode arranca, crea una imagen nueva resultado de la unión de la última imagen almacenada y el log de cambios registrados en ella. A medida que los DataNodes escriben en el HDFS, van enviando las modificaciones pertinentes al NameNode que mantiene actualizados los cambios en el log, pero no modifica la imagen creada al arrancar. El despliegue típico del NameNode, para que la escritura del log sea segura, incluye actualizaciones de las copias del *log* en el sistema de ficheros local, en la memoria del NameNode y en un NFS remoto.

Checkpoint Node

El fin que se persigue agregando un nodo de este tipo es mitigar los problemas relacionados con la caída de operación del NameNode. Periódicamente, el Nodo de Checkpoint va generando puntos de restauración, o *checkpoints*, siguiendo la misma estrategia que en el NameNode, es decir, usando `fsimage` y `edits`. Cada cierto tiempo, se descargarán ambos ficheros desde el NameNode y se fundirán, formando una nueva imagen actualizada que será transferida al NameNode. Cuando haya finalizado con éxito la operación, el NameNode tendrá que purgar la imagen antigua e iniciar un nuevo fichero de log que albergue los cambios que se vayan sucediendo.

Backup Node

El Nodo de Backup provee la misma funcionalidad de generación de puntos de restauración que el Checkpoint Node pero usando una aproximación diferente. Para mantener sincronizado su espacio de nombres con el del NameNode que plagia, este tipo de nodo se descarga el `fsimage` al arrancarse y lo actualiza con las modificaciones que vaya captando el NameNode. Adicionalmente y cada cierto tiempo, el Backup Node actualiza su `fsimage` con los `edits`, creando un punto de restauración asociado.

En comparación con el Checkpoint Node, este nodo consume menos ancho de banda de red ya que no necesita descargarse el `fsimage` y los `edits` del NameNode para mantener el sincronismo de estado.

Como apuntes finales, destacar que de momento (versión 1.0.4 de Hadoop), sólo se soporta un Nodo de Backup por NameNode o múltiples Nodos de Checkpoint, y que la presencia de un Backup Node habilita la posibilidad de correr el NameNode sin almacenamiento persistente, delegando esa responsabilidad al Nodo de Backup.

5.3.2 Topología de red

Una de las partes fundamentales de un sistema de ficheros en un entorno distribuido es proveer al usuario de un mecanismo transparente, que garantice la persistencia de la información en él contenida manteniendo un cierto nivel de prestaciones. El HDFS utiliza una técnica, ya citada para los cloud, la replicación, que proporciona altas prestaciones, escalabilidad y tolerancia a fallo, al tiempo que limita la congestión de red controlando la ubicación y el número de copias de los bloques en el centro de datos.

Distancia entre nodos

Para poder soportar las prestaciones comentadas, es fundamental que el NameNode, que es quien gestiona la distribución de las réplicas de los bloques, tenga cierto conocimiento de la organización física de los nodos que participan en el despliegue. La idea fundamental es tratar de mantener un equilibrio en la separación de las copias, entendida como la *distancia física* entre los nodos que almacenan cada una: la distancia media entre réplicas es proporcional tanto a la tolerancia a fallo del sistema, como al ancho de banda consumido para enviar cada réplica. Recordemos que el ancho de banda de red disponible para transferir información entre distintos nodos se reduce a medida que los alejamos, o visto de desde otro ángulo, que la transferencia será más costosa cuanto mayor sea la distancia entre el nodo que contenga el bloque original y aquel que vaya a albergar la réplica. Es decir, sería idóneo, para equilibrar la separación entre réplicas, definir una métrica que calculase la distancia entre dos nodos cualesquiera.

Como los nodos de las redes IP siguen una estructura de árbol invertido, y la red de un clúster Hadoop es de este tipo, se podría considerar, como aproximación de la distancia física entre nodos, la `distancia internodal`: *la suma de las distancias de los nodos al ancestro común más próximo.*

Tal y como se ha comentado, HDFS utiliza RPC sobre TCP/IP a través de SSH para la comunicación entre nodos, lo cual (capa IP) no aporta información de localización *concreta* de cada nodo dentro del despliegue. Para concretar la distancia internodal y así poder hacer un reparto óptimo de las copias, es necesario configurar HDFS para que cada IP sea mapeada a una posición concreta, de tantas componentes —`(centro de datos, rack, nodo)`, por ejemplo— como niveles haya en la red. La figura 5.4 muestra un ejemplo con los valores más comunes de la distancia internodal. Veremos cómo se calcula para el caso `d=4`.

Fijándonos en la figura 5.4, el bloque azul representa la copia original, y el bloque amarillo, destino del arco azul con etiqueta `d=4`, la réplica. Ambos bloques se encuentran en nodos de distintos racks en el mismo centro de datos. El ancestro común más próximo entre ellos será el enrutador que maneje el tráfico entre ambos racks. Además, cada nodo tiene que atravesar otro enrutador de rack, otro ancestro más próximo a los nodos, que dirige la paquetería dentro del rack. Con lo que tenemos *dos pasos* (distancia 2) para llegar al router que conecta ambos racks por cada nodo; sumando ambas distancias obtenemos el resultado.

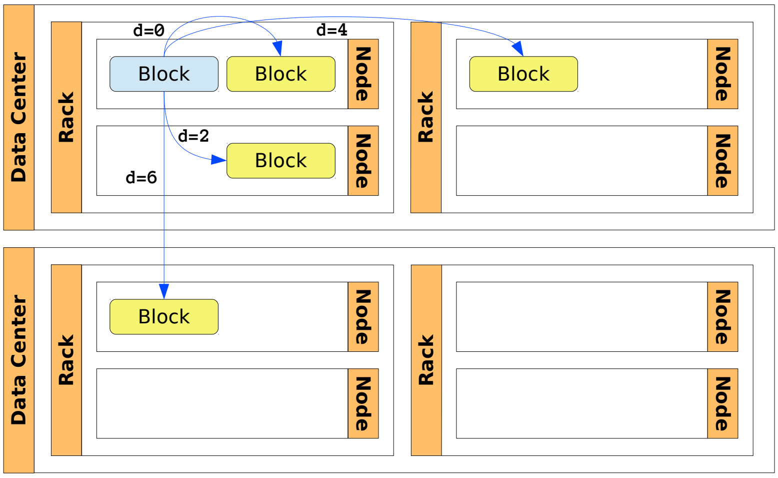

Figure 5.4: Ejemplo de valores de distancias internodales

Replicación

La replicación es una técnica transparente al usuario y controlada por el NameNode que, para ser explotado en óptimas condiciones, debería tener conocimiento de la distancia internodal. El grado de separación entre dos réplicas es directamente proporcional al grado de robustez que aporta la copia —la tolerancia a fallo— e inversamente proporcional a la eficiencia de transmisión por red, puesto que el ancho de banda disponible para mover un bloque entre dos centros de datos, será menor que para gestionar la réplica de modo local al nodo, como ya hemos indicado. De tal manera que si se produjese la caída de un computador, la probabilidad de propagación de esa caída disminuye a medida que nos alejamos del nodo problemático —imaginemos una inundación del centro de datos, por ejemplo.

La estrategia concreta que sigue HDFS consiste en colocar la primera réplica en el mismo nodo que el cliente del sistema de ficheros, si éste pertenece al clúster de almacenamiento —una aplicación cliente corriendo en un nodo del clúster HDFS, por ejemplo. En caso de que el cliente sea externo al clúster, se elige un nodo al azar, teniendo en cuenta su carga computacional

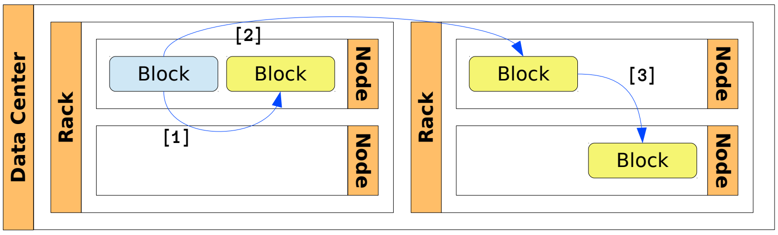

Figure 5.5: Ejemplo de replicación de un bloque, factor 3

—priorizando los de menor carga. La segunda réplica se coloca fuera del rack en el que se encuentre la primera copia; el rack concreto se elige al azar. La tercera se emplaza en el mismo rack que la segunda copia pero en un nodo diferente, de nuevo eligiendo al azar y balanceando carga. Las réplicas sucesivas —el factor de replicación se controla en el fichero de despliegue de HDFS— se envían a nodos, siempre diferentes, de este último rack.

La mecánica descrita aporta el equilibrio deseable entre tolerancia a fallo (ya que habrá copias de cada bloque en dos racks distintos), ancho de banda consumido (ya que la escritura de cada bloque sólo atraviesa un *switch* o enrutador y las copias sucesivas se hacen dentro del mismo rack), rendimiento de lectura (al poder elegir entre dos racks ante cada petición de lectura de un bloque) y distribución equilibrada de los bloques en el clúster (que realiza HDFS al ejecutar el método descrito). La figura 5.5 muestra un ejemplo de replicación con factor 3 en un solo centro de datos. El número entre corchetes indica el orden de creación de cada copia; el bloque original, recientemente actualizado o creado, es el azul.

5.4 Hadoop MapReduce

En un clúster típico sobre HDFS se dispone la capa de operación de Hadoop, Hadoop MapReduce, que lleva a cabo la ejecución de los trabajos de mapeo y reducción enviados al framework. Habiendo ya introducido las bases del

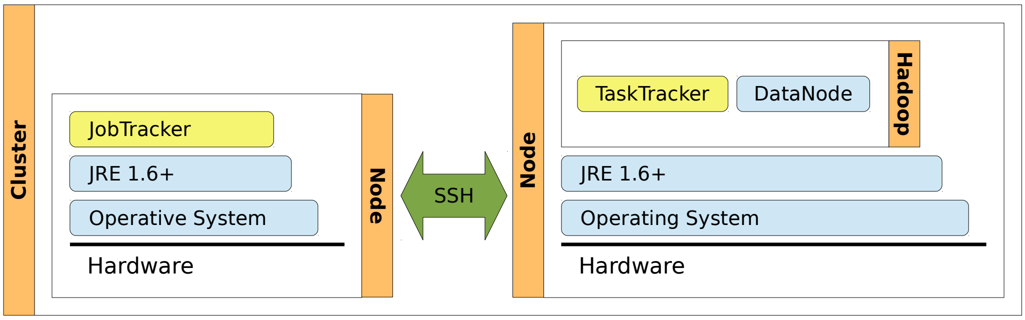

Figure 5.6: Ejemplo de despliegue de Hadoop MapReduce

paradigma MapReduce, corresponde ahora especificar aquellas desviaciones concretas de la implementación en Hadoop. Como era esperable, Hadoop MapReduce se basa en los principios expuestos en el artículo de Google [2] en cuanto a reparto de tareas y gestión de fallos. Su arquitectura de alto nivel no dista mucho de la observada en la capa HDFS y así se distinguen los roles *maestro* y *esclavo*.

Por una parte, aparece el *JobTracker*, encargado de planificar el reparto de las tareas a los nodos del clúster; por la otra, el *TaskTracker*, que ejecuta las tareas en los nodos como funciones Map y Reduce.

Igual que su sistema de ficheros distribuido, Hadoop MapReduce está escrito en Java.

5.4.1 Roles de los nodos

La figura 5.6 representa un despliegue típico de Hadoop MapReduce en un clúster, utilizando HDFS como sistema de ficheros distribuido soporte. Para completar la *fotografía* habría que incluir los nodos responsables de gestionar el HDFS —NameNode, Checkpoint Node y Backup Node— omitidos por claridad.

JobTracker

El comportamiento del nodo JobTracker es muy similar al expuesto en el caso del NameNode de HDFS, pero aplicado a la gestión de trabajos y tareas. Ante una petición de ejecución, el JobTracker dividirá el trabajo asociado en tareas que repartirá entre los TaskTrackers que tenga bajo supervisión. Normalmente, el tamaño de los ficheros de entrada de las tareas se hace coincidir con el de los bloques del sistema de ficheros distribuido, sea o no HDFS, por ser lo más eficaz. Para cerciorarse de que las ejecuciones concluyen con éxito en un entorno expuesto al fallo, el JobTracker mantiene una lista de estado de las tareas asociadas a los nodos TaskTracker. De tal forma, en caso de que se produjese un error que impidiese la finalización de alguna tarea, ya sea una tarea Map o una Reduce, el JobTracker replanificaría su ejecución en otro nodo disponible con la mínima carga computacional.

El reparto de trabajo se lleva a cabo siguiendo la máxima localidad, esto es, haciendo que los TaskTrackers reduzcan tareas basadas en la transformación de datos almacenados en el mismo nodo. Así se reducen tanto el tiempo de acceso al dato del TaskTracker, como la saturación de la red del clúster, haciendo la computación más ligera.

Tal y como sucedía en la definición general del MapReduce de Google [2], el fallo en un JobTracker es muy problemático porque sólo está cubierta la posibilidad de correr uno por clúster sin usar herramientas adicionales. Dada una caída en el JobTracker, el procedimiento de recuperación “resuelve” la situación descartando los trabajos sin concluir; esperando que el nuevo JobTracker pueda hacerse cargo del procesado de esos trabajos incompletos que habrán de ser enviados manualmente por los usuarios. Actualmente sí se pueden manejar JobTrackers adicionales en un mismo clúster e instante temporal usando una herramienta adicional: *Zookeeper*.

TaskTracker

La misión fundamental del TaskTracker es procesar las tareas que le sean enviadas desde el JobTracker. Periódicamente, el TaskTracker envía una señal a su JobTracker para informar acerca del estado de progreso de la ejecución de una tarea, si tuviese una asignada, o para indicar que se encuentra a la espera. Si el JobTracker no recibiese esa señal en un intervalo convenido, éste marcaría el TaskTracker y *todas* sus tareas relacionadas —las concluidas y las incompletas— como inaccesibles. Esta clase de fallo, es decir la caída de un TaskTracker, se considera menos problemático que en el caso del JobTracker, pero implica replanificación de tareas e incluso repetición de la ejecución de alguna. Usando la comentada lista de estado de las tareas, el JobTracker buscará las inaccesibles necesarias —las Map completadas y las Reduce cuya salida no esté en el DFS— y hará la redistribución siguiendo la mecánica descrita.

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`. Accedido: junio 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/`. Accedido: junio 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/`. Accedido: junio 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/`. Accedido: junio 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. Accedido: junio 2013.

[19] How to Use CloudStack without Hardware Virtualization. `http://support.citrix.com/article/CTX132015`, 2012. Accedido: junio 2013.

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

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

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

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

[24] Getting Started with Openstack on Fedora 17. `http://fedoraproject.org/wiki/Getting_started_with_OpenStack_on_Fedora_17`, 2012. Accedido: junio 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. Accedido: junio 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. Accedido: junio 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] Jacek Artymiak. *Programming OpenStack Compute API.* Rackspace US Inc., 2012.

[39] *OpenStack API Reference*, 2013.

[40] OpenNebula OCCI Specification 3.8. http://opennebula.org/documentation:archives:rel3.8:occidd, 2012. Accedido: junio 2013.

[41] DEISA. Deisa glossary. http://www.deisa.eu/references. Accedido: junio 2013.

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

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

[44] Simon Kelley. dnsmasq — A Lightweight DHCP and Caching DNS Server — ManPage. http://www.thekelleys.org.uk/dnsmasq/docs/dnsmasq-man.html, 2012. Accedido: junio 2013.

[45] HDFS Users' Guide. http://hadoop.apache.org/docs/r1.0.4/single_node_setup.html, 2010. Accedido: junio 2013.

[46] Single Node Setup. http://hadoop.apache.org/docs/r1.0.4/single_node_setup.html, 2008. Accedido: junio 2013.

[47] MapReduce Tutorial. http://hadoop.apache.org/docs/r1.0.4/mapred_tutorial.html, 2008. Accedido: junio 2013.

[48] Cluster setup. http://hadoop.apache.org/docs/r1.0.4/hdfs_user_guide.html, 2008. Accedido: junio 2013.

[49] Dhruba Borthakur. HDFS Architecture Guide. http://hadoop.apache.org/docs/r1.0.4/hdfs_design.html, 2012. Accedido: junio 2013.

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

[51] 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.

[52] 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.

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

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

[55] Bruce Barnett. Sed — An Introduction and Tutorial. `http://www.grymoire.com/Unix/Sed.html`, 2012. Accedido: junio 2013.

[56] MapReduce Wikipedia entry. `http://en.wikipedia.org/wiki/MapReduce`, 2012. Accedido: junio 2013.