Assessment, design and implementation of a private cloud for MapReduce applications

Patricia González Gómez
Department of Electronics and
Systems at UDC
Campus de Elviña s/n, 15007
A Coruña, Spain
patricia.gonzalez@udc.es

José Carlos Cabaleiro
Domínguez
Institute for Clarity in
Documentation
P.O. Box 1212
Dublin, Ohio 43017-6221
jc.cabaleiro@usc.es

Tomás Fernández Pena
The Thørväld Group
1 Thørväld Circle
Hekla, Iceland
tf.pena@usc.es

Marcos Salgueiro Balsa
Improving Metrics
Brookhaven National Lab
P.O. Box 5000
marcos.salgueiro@gmail.com

ABSTRACT

The extraordinarily vast amount of information generated as a byproduct of Internet usage, has been embodying an increasing burden to traditional procedures and models, unable to handle it efficiently due to its heterogeneous nature. Besides, as the volume of information grows so does the size of the datacenter required to process and store it, quickly overloading its full capacity when demand peaks. Together —*not relational* data and uneven demand distribution— they shape the basis of modern data-driven request servicing.

A series of technologies have been developing lately to manage this scenario. Two of the most highlighted among them are *MapReduce* and *Cloud Computing*. *MapReduce* was introduced in [4] to abstract the common difficulties linked to distributed processing on large clusters. *Cloud Computing*, on the other hand, agglutinates miscellaneous subsystems forming a unified interface to flexibly deploy and manage virtual clusters.

This paper explores their potential symbiosis, in order to create a robust and scalable environment, to execute *MapReduce* workflows regardless of the underlaying infrastructure. It also details a proof of concept implementation using open source tools, similar to Amazon's own *Elastic MapReduce*.

Keywords

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

WOODSTOCK '97 El Paso, Texas USA

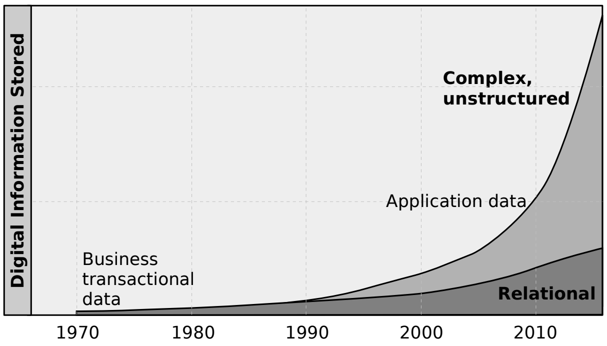

Figure 1: Unstructured and relational data volume evolution. Source: *Cloudera Inc.*

1. INTRODUCTION

The proliferation of Internet-enabled handhelds and the continuously improving access speed, have set a background in which user services are becoming heftier —from SQ Video yesterday to HD today and 4K tomorrow—, are being consumed throughout the day and are requiring an increasing amount of user-related data —GPS position, locale, personal settings, filters, previous searches or purchases, connections, friends, retweets, etc.— to take into account. It is this last trait what have been representing the biggest trouble: the *class* of data packed within these services cannot be modeled by traditional standards, as it lacks a relational structure.

While some argue that every miniworld may be *transformed* into a Relational Model, it is the necessity to lay out the data structure before information can be saved and put to use what poses a central obstacle in making these models adapt to such a swiftly mutating data. As figure 1 depicts, the gap between relational and unstructured information continues to widen, that is why there has been an explicit push off schema-driven modeling towards loosely-structured representations.

So, relaying on *schemaless* data definition allows to better cope with unstructured information. There are still, however, two dimensions to discuss: data volume and nonuniform access distribution.

To handle data flowing in at Internet scale there has to be devised a distributed processing model beyond large clusters, high capacity networks and intelligent load balancers. To deal with that sea of data, *MapReduce* processing model splits input all the way down to unrelated pairs of *unique* key and key-related data. Using the approach to uniquely identify each *atomic* piece of information, allows to easily apply a fair distribution policy across participating nodes able to reduce network transfers and to recover from failure.

Finally, clusters' capacity has to be able to accommodate a variable number of information requests per second, reducing idle node time without implying a loss in service quality. An ideally suited technique to that end is *Cloud Computing*. *Cloud Computing* has been making headlines as of late praised for its inherent nature to scale-out virtual deployments effortlessly, and so, capable of stretching and shrinking computational power with demand needs.

Inasmuch as *MapReduce* and *Cloud Computing* together may prove useful in servicing a potential world of data consumers, it is easy to understand the growing interest in both technologies. Currently, the best known example of a unified approach to said technologies is *Amazon Elastic MapReduce* (EMR) [1]. Nonetheless, there are other implementations focusing on extending EMR's functionality, either by surpassing its constraints —information must be made semi-public and *MapReduce* workflows need to be executed on Amazon's installation— with *Resilin* [7], *Savanna* [3] or *Dynamic MapReduce* [6], or by reusing its cloud interface to build a *MapReduce* platform upon like with *Cloud MapReduce* [5].

The major contribution of this work is a simple and unified interface to manage *MapReduce* computations, leveraging any existing *IaaS* deployment with a little customization, while providing an automatic one node test installation based on *OpenStack* and *Apache Hadoop*. We have called our implementation *qosh* and it has been written in *Python*.

Section 2 elaborates on current IaaS framework implementations, focusing on its highlights and drawbacks. Section 3 details *qosh*'s architecture and self-installing deployment structure. Section **??** goes through a complete execution cycle, identifying key points of the process. Section 4 reviews qosh performance when deployed on a real cluster. Section 5 collects a reflection on *qosh*'s main contributions and future development guidelines.

2. IAAS CLOUD FRAMEWORKS

In order to abstract virtual cluster creation and destruction *qosh* relies on an *IaaS cloud* framework. Even though there is a good number of them, they all share a common architecture and cover a similar set of functionality with mixed maturity levels.

Structurally, they are comprised of a series of modules connected together by an asynchronous message broker. Internally, they save their processing information in a database and exploit their server's hardware through the use of a hypervisor; externally, they expose their capabilities implementing a REST interface to be consumed by a demanding client.

It could be argued that any listing that covers the most widely used IaaS cloud frameworks must include *OpenStack*, *CloudStack*, *OpenNebula* or *Eucalyptus*. Precisely, in order to determine which of them could couple *qosh* best, a reduced installation was carried out before putting them to use.

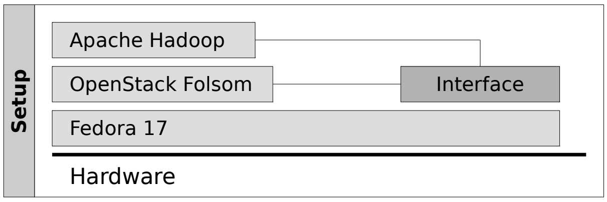

Figure 2: High level design diagram

Our testing methodology considered diverse subjective magnitudes —documentation completeness, installation complexity, modular flexibility, standardization, etc.— to give a general view of each one of them. In the end, *OpenStack* came up on top inasmuch as the latest two releases have immensely improved both its reach in real deployments and its perceived functional maturity.

3. ARCHITECTURE AND IMPLEMENTATION

qosh's setup defaults to a single node installation in which both infrastructure and execution environment are configured. Figure 2 precisely depicts the layered configuration. Atop Fedora 17 our setup script downloads and installs *OpenStack* precompiled packages, and afterwards it downloads, untars and registers a virtual machine image containing an *Oracle 1.7 JRE* and *Apache Hadoop 1.0.4* installation. Likewise, it automatically creates the right user and tenant so that *qosh* may be put to use straightaway.

At the right end of Figure 2, it appears an *Interface* module lying on top of Fedora and being connected to both *OpenStack* and *Apache Hadoop*. Its main purpose is to deploy virtual *Hadoop* clusters, to manage its component virtual machines' —or *VM*s— lifecycles and to orchestrate *MapReduce* workflows executions.

3.1 Initial setup

qosh's own installation script will automatically configure a highly-performing testing environment that could be easily scaled-out as demand grows. Figure 3 represents the layered setup decomposition in a single node after the installation procedure had finished.

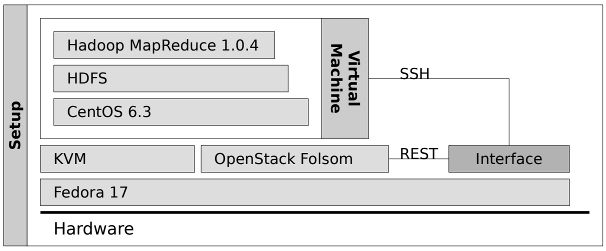

Figure 3: Layered initial deployment

The *OpenStack* modules deployed are those fundamentally required by a minimum standalone setup:

Keystone manages authorization, authentication and quota by user and *tenant*.

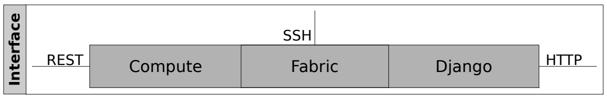

Figure 4: Interface composition

Nova handles VMs' lifecycles and networking configuration, routing and data flow utilizing the *Kernel Virtual Machine* (KVM) as hypervisor.

Glance holds the browsable catalog of installed VM images on the local file system.

Which implies that no fault tolerance measures are defined —as expected from a single node and local file system arrangement— cloud-wise, but it certainly allows for other standard safety protocols to be implemented —on the order of some RAID level with replication or UPS solutions.

3.2 Interface

Figure 4 represents the user interface's modular composition. There are three essential modules within:

3.2.1 Compute

Compute is the REST access client that bridges the *OpenStack* cloud with the web interface, effectively decoupling *qosh* from the infrastructure provider. It basically encapsulates a series of methods by which an authorized user is allowed to manually define VM deployment behavior.

Current implementation manages virtual clusters defined with *OpenStack* running on a single real cluster, i. e. no hybrid clouds are supported. However, *Compute* may be effortlessly adapted to handle VMs running on other IaaS deployments or to manage hybrid clouds, with no interaction whatsoever with another module, as far as *qosh* API semantics are preserved.

3.2.2 Fabric

Fabric is a *Python* library used to simplify managing our virtual cluster by establishing SSH tunnels with the VMs, letting *qosh* shape *Hadoop* configuration, put processing data into HDFS —Hadoop Distributed File System— and recover results to user space; everything as SSH traffic.

To establish SSH connections our *Fabric* module is fed a *Keystone*-generated keypair. This keypair is created on each virtual cluster deployment and shared by all VMs in the same cluster. Its private part is injected into VMs once they have finished booting (refer to section **??** for an implementation), and its public part is kept on the local file system. It is automatically removed —both from *OpenStack* and file system— when *Apache Hadoop* execution completes.

3.2.3 Django

Django glues together both modules, renders HTML to be displayed to the user and organizes result and metadata storage.

Django can be plugged different back-ends, from session objects managers to static file storage, to deal with varying needs and to accommodate future demands. *qosh*'s plugin configuration includes: *MySQL*, used as meta-information repository; the server file system, to save and retrieve *MapReduce* I/O data and *OpenStackBackend* to delegate into *Keystone* user access and quota.

Putting it all together, a user would define *MapReduce* computations though a Django-backed web interface. Django would pass configuration parameters on to *Fabric* for creating and feeding input data to a virtual *Hadoop* cluster. And lastly, real infrastructure would be provisioned by an IaaS cloud driven through *Compute* module.

3.3 Deployment

qosh's installation script will take care of a single node deployment in an automatic fashion, so no previous knowledge of *OpenStack* or *Hadoop* would be required to exploit *qosh*'s elastic *MapReduce* prowess in this case; though the virtual cluster's *elasticity* would be heavily constrained. To overcome this limitation, *qosh* has been architected to abstract the infrastructure underneath, allowing for any IaaS framework to be deployed at any size —some *Compute* module's parts would require rewriting, nonetheless, if *OpenStack* were not used.

An installation may be grown from a starting single node setup just by laying out a real IaaS cloud cluster of any size. In fact, any public cloud *Amazon Elastic Compute Cloud*-compatible (EC2-compatible) could be used to expose infrastructure that *qosh* would utilize to spawn virtual clusters of any size.

3.4 Apache Hadoop Virtual Machine

The *Apache Hadoop* installation has been manually configured from scratch inside a virtual machine. It's been conceived to have a minimum footprint while maintaining a server-grade stability. In order to fulfill these requirements *CentOS* was chosen and an EC2-compatible VM was built up with it.

An EC2-compatible VM differs from a *regular* VM in a few peculiarities:

Container format is really subject to framework requirements but the most commonly preferred formats are raw and qcow2 for cloud images, while traditional VMs depend exclusively on the virtualization platform support.

User access is controlled by injecting a private part of an SSH keypair into booting VMs, so that only users with the public counterpart are allowed to log in. However, that private part is not *pushed* into the VM file system by the cloud framework itself, it is *pulled* instead to a web location, concealed from other VMs, so it is the VM's duty to fetch and safeguard that keypair.

VHDD resizing which is the ability to change the VM's HDD size on demand, can only be accomplished if *direct kernel boot* was being used. Enabling a VM image to boot a kernel directly implies extracting both initram and kernel images from the VM file system and uploading them to the particular cloud framework deployed.

Bearing those singularities in mind, an *Apache Hadoop* and *Oracle JRE* installations, a limit in what kind of SSH connections can be established —only those authenticated by keypair— and a final compression, together, yielded *qosh*'s VM [2] which has potential to be executed on any cloud distribution —considering it being EC2-compatible.

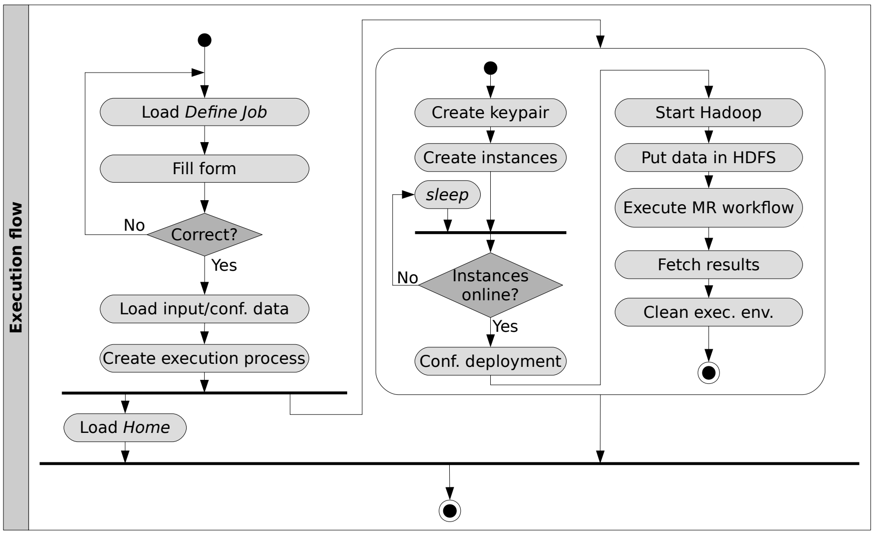

Figure 5: Global execution flow

3.5 Execution flow

Before any MapReduce processing take place, a user should log in and navigate to *Define Job* page. Figure 5 contains a visual representation of a complete execution cycle starting up from that point.

- When *Define Job* is completely rendered, the user is presented a form to configure a new MapReduce job and its supporting virtual cluster.

- In case the form be correctly filled, all of the input data and configuration parameters would be uploaded server-side.

- Once the upload have finished, a new process is spawned to manage the remaining procedure; meanwhile, the user is sent back to the *Home* page.

- To guarantee a fair level of privacy, an SSH keypair is created anew on each MapReduce execution. Along with it, a set of virtual machines, or instances, is started.

- As the amount of time required to bring up networking on each instance varies depending on virtual and real cluster size, a mechanism to check their networking status had to be devised. In order to reduce the complexity and coupling introduced by making the instances fire a *networking-ready* signal, we came up with a better option. Instead of pushing a *ready* event from the VM to the cloud, the process supervising their creation is kept looping trying to establish an SSH connection to the instances, up to a certain number of attempts. To reduce CPU overload, a one second delay is set between retries.

- Once every instance can be reached through SSH, a virtual *Hadoop* cluster is configured following the guidelines contained in *Fabric*'s script.

- Through Fabric mediation, *Hadoop* daemons are started on every instance, input data and workflow implementation are pushed onto HDFS and the MapReduce application is started.

- When the job be finished, the results will be fetched from the virtual cluster to the local file system, where they will be permanently stored.

- Lastly, the instance set is destroyed and the keypair removed from both *OpenStack* and the local file system.

4. PERFORMANCE EVALUATION

To assess a measure on *qosh*'s performance a custom deployment was carried out on two nodes. In one of them, the automatic installation procedure was executed first and the resulting configuration tweaked later to communicate with the other node. In the other, the bare minimum required to allow for VM execution was manually installed —*OpenStack Compute* and required libraries.

Both machines share the same physical configuration which is comprised of an octo-core Intel Xeon CPU layout, 8 GB RAM, 200 GB SATA 3 HDD and dual Gigabit Ethernet connectivity.

4.1 Testing methodology

In order to give a general picture of *qosh*'s performance we set up three different testing cases in which we measured the time required by the whole executing process. We also *hard coded* timing marks to cut relevant parts off and to reduce measurement errors. To that account —to bound the experiment variance—, testing cases were executed and measured ten times per deployment configuration; displayed results reflect a simple average of said measures.

Five components were gauged:

Deploying time stands for the time interval elapsed from the instance the virtual cluster is starting to be spawned up to when all of the instances can be reached. It should be noted this component adds up to a second of error per instance, due precisely to the one second delay between retries as explained in section 3.5.

Configuring time alludes to *Hadoop* configuration time requirement. It includes transferring and decompressing input data, pushing them to HDFS and laying *Hadoop* configuration out in the virtual cluster.

***MapReducing* time** covers exclusively the amount required to complete the *MapReduce* job.

Cleaning time spans just the temporal lapse that is needed to remove the keypair and to shutdown every instance in the cluster.

Total time is the time it took the whole process to complete. Note that it does not equal partial times summation.

The *MapReduce* application that we used is the *well-known* `Wordcount`, which counts the number of words in an actual document set. *Apache Hadoop* was configured to allocate to its underlaying JVM up to the maximum RAM available to the instance.

The first test case centered on evaluating *qosh*'s timings when the virtual cluster was scaled out. Thus, a fixed 62.5 MB plain text document list was fed to a growing virtual cluster ranging from one 1 GB RAM/VCPU instance to eight —4 instances on each node, as *OpenStack* was laid over a dual-node cluster.

For the second test case, the number of VMs spawned remained constant in different executions but its capabilities —RAM and VCPU count— were sequentially doubled from 1 GB RAM/VCPU to reach 4 GB RAM/VCPU, to account for *qosh*'s tendency on vertical scaling. The 62.5 MB of plain text was kept untouched as on the first case.

Lastly, a third test case presents the resulting measurements obtained on the same virtual cluster configuration, when the input size was sequentially doubled from 62.5 to 250 MB.

4.2 Results analysis

Figure 6 presents the evolution of different timings as instance count increases from one to eight. It can be seen how deploying, configuring and clearing measurements rise slightly with instance count, while processing time drops, all as expected.

Figure 7

5. CONCLUSIONS AND FUTURE WORK

6. REFERENCES

[1] Amazon Web Services: Elastic MapReduce. `http://aws.amazon.com/elasticmapreduce/`, Oct. 2013.

[2] Apache Hadoop 1.0.4 based on CentOS 6.3 VM. `http://drive.google.com/.......................`, Oct. 2013.

[3] OpenStack: Project Savanna. `https://wiki.openstack.org/wiki/Savanna`, Oct. 2013.

[4] J. Dean and S. Ghemawat. Mapreduce: simplified data processing on large clusters. In *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, OSDI'04, pages 10–10, Berkeley, CA, USA, 2004. USENIX Association.

[5] H. Liu and D. Orban. Cloud MapReduce: A MapReduce Implementation on Top of a Cloud

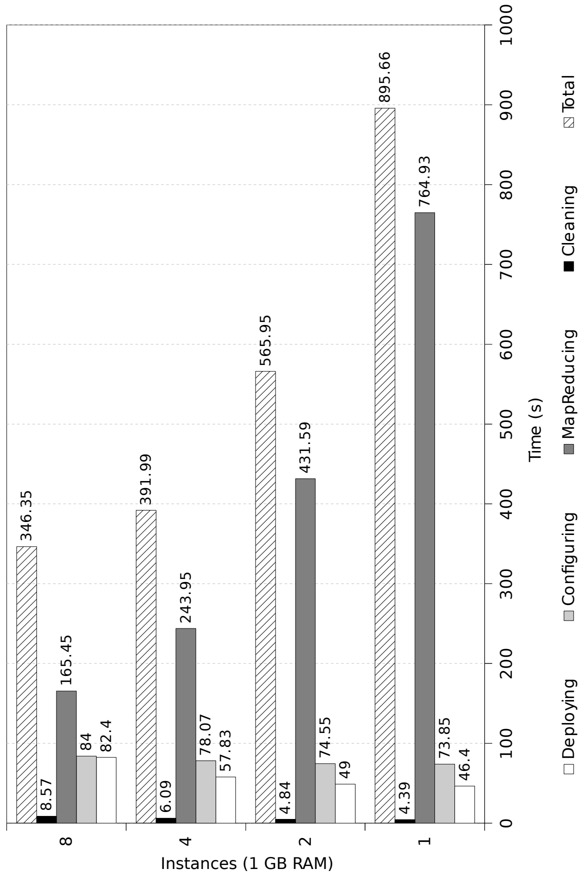

Figure 6: Tendency on scaling out

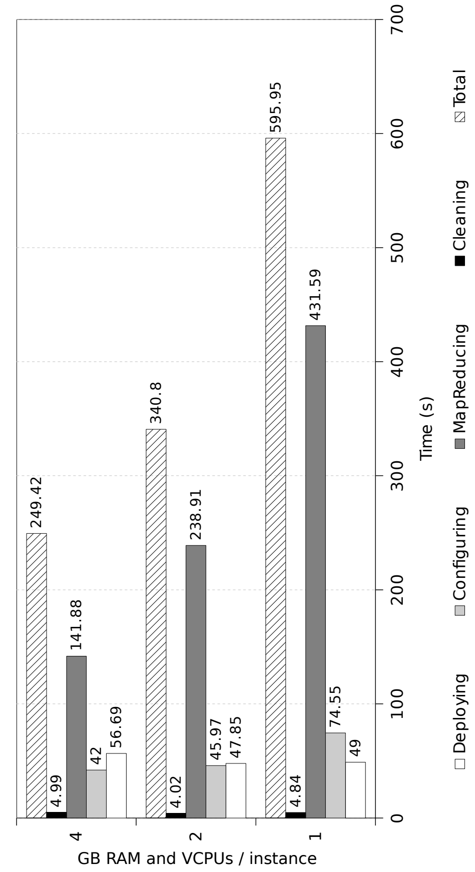

Figure 7: Tendency on scaling in

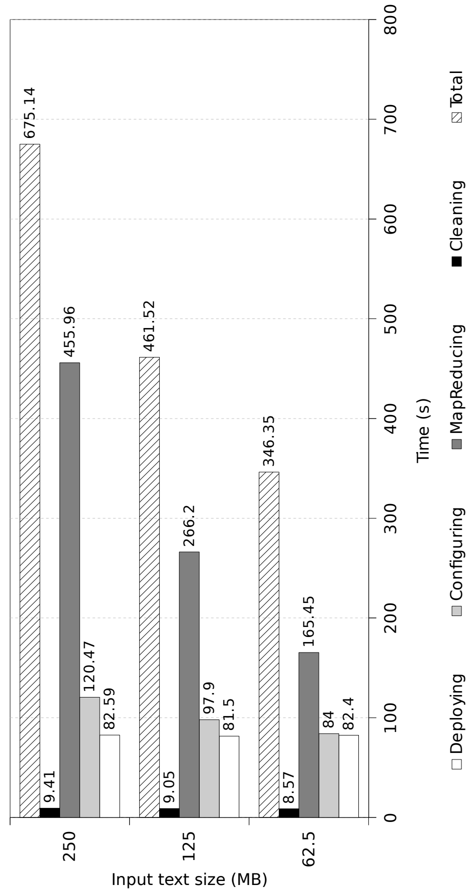

Figure 8: Tendency on input size escalation

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.

[6] S. Loughran, J. M. Alcaraz Calero, A. Farrell, J. Kirschnick, and J. Guijarro. Dynamic Cloud Deployment of a MapReduce Architecture. *IEEE Internet Computing*, 16(6):40–50, Nov. 2012.

[7] P. Riteau, A. Iordache, and C. Morin. Resilin: Elastic MapReduce for Private and Community Clouds. Research Report RR-7767, INRIA, Oct. 2011.

[8] T. White. *Hadoop, the Definitive Guide.* O' Reilly and Yahoo! Press, 2012.