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

Marcos Salgueiro*, Patricia González*, Tomás F. Pena† and José C. Cabaleiro†
*Department of Electronics and Systems
Univ. of A Coruña, A Coruña, Spain
Email: marcos.salgueiro@gmail.com, patricia.gonzalez@udc.es
†Centro de Investigacións en Tecnoloxías da Información, CITIUS
Univ. of Santiago de Compostela, Santiago de Compostela, Spain
Email: tf.pena@usc.es, jc.cabaleiro@usc.es

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 developed lately to manage this scenario. Two of the most highlighted among them are *MapReduce* and *Cloud Computing*. MapReduce was introduced in [1] 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 underlying infrastructure (*Hadoop as a Service*, HaaS). It also details a proof of concept implementation using open source tools, similar to Amazon Elastic MapReduce.

Keywords-MapReduce, Hadoop as a Service, OpenStack, Private clouds, Cloud Computing, Virtualization.

I. 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.
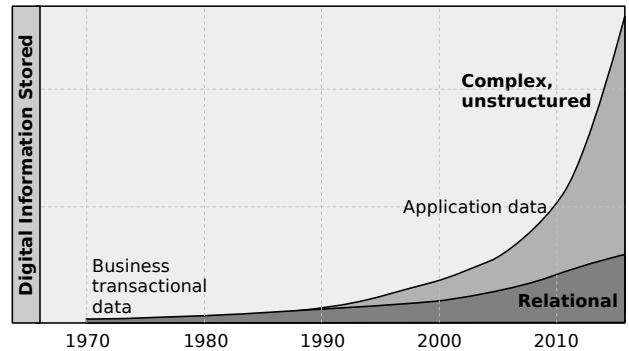

Fig. 1. Unstructured and relational data volume evolution. Source: *Cloudera Inc.*

So, relaying on schemaless data definition allows to better cope with unstructured information. There are still, however, two dimensions to discuss: data volume and non-uniform 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 reach that objective 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) [2]. Nonetheless, there are other implementations focusing on extending EMR 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* [3], *Savanna* [4] or *Dynamic MapReduce* [5], or by reusing its cloud interface to build a MapReduce platform upon like with *Cloud MapReduce* [6].

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* (*quick-openstacked-hadoop*) and it has been written in Python.

Section II elaborates on current IaaS framework implementations, focusing on its highlights and drawbacks. Section III details qosh architecture and self-installing deployment structure, and goes through a complete execution cycle, identifying key points of the process. Section IV reviews qosh performance when deployed on a real cluster. Finally, section V collects a reflection on qosh main contributions and future development guidelines.

II. IaaS cloud frameworks

In order to abstract virtual cluster creation and destruction qosh relies on an *IaaS cloud* framework. Even though there are 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 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* [7], *CloudStack* [8], *OpenNebula* [9] or *Eucalyptus* [10]. Precisely, in order to determine which of them could couple qosh best, a reduced installation was carried out before putting them to use.

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.

III. Architecture and implementation

qosh 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
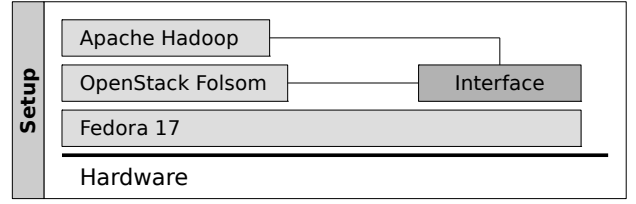

Fig. 2.   High level design diagram

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 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.

A. Initial setup

The qosh 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.
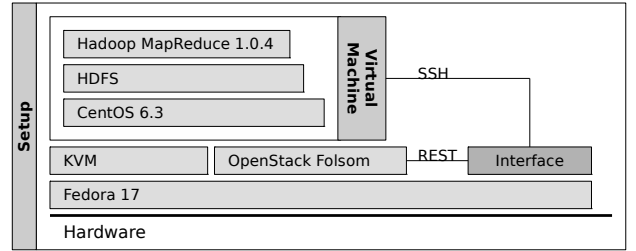

Fig. 3.   Layered initial deployment

The OpenStack modules deployed are those fundamentally required by a minimum standalone setup [7]:

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

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

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

This 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 —e.g. some RAID level with replication or UPS solutions.

B. Interface

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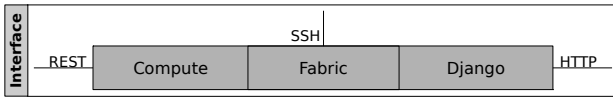
Fig. 4. Interface composition

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.

2) Fabric: *Fabric* is a Python library used to simplify the management of our virtual cluster by establishing SSH tunnels with the VMs, letting qosh shape Hadoop configuration, putting processing data into HDFS (Hadoop Distributed File System) and recovering 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, and its public part is kept on the local file system. It is automatically removed —both from OpenStack and file system— when Hadoop execution completes.

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. The qosh 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.

C. Deployment

The qosh 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 elastic MapReduce prowess in this case; though the virtual cluster 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 parts of the *Compute* module 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.

D. Apache Hadoop Virtual Machine

The Apache Hadoop installation has been manually configured from scratch inside a virtual machine. It has 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 VMs duty to fetch and safeguard that keypair.
- VHDD resizing: which is the ability to change, on demand, the HDD size of the VMs, 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 V [11] which has potential to be executed on any cloud distribution (considering it being EC2–compatible).

E. Execution flow

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

1) When *Define Job* is completely rendered, the user is presented a form to configure a new MapReduce job and its supporting virtual cluster.
2) In case the form be correctly filled, all of the input data and configuration parameters would be uploaded server-side.
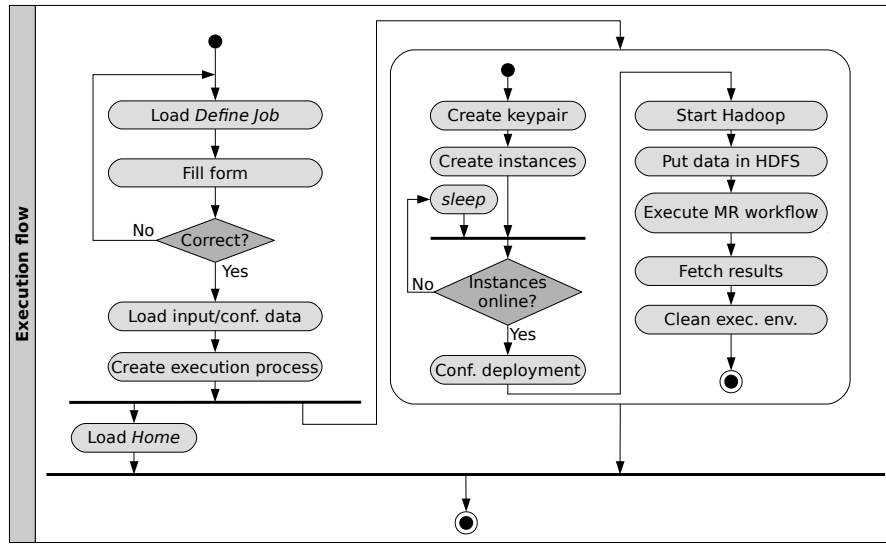
Fig. 5. Global execution flow

3) 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.

4) 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.

5) 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.

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

7) 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.

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

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

IV. Performance evaluation

To assess a measure on qosh 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.

A. 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 instant that 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 III-E.
- *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 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 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.

B. 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.
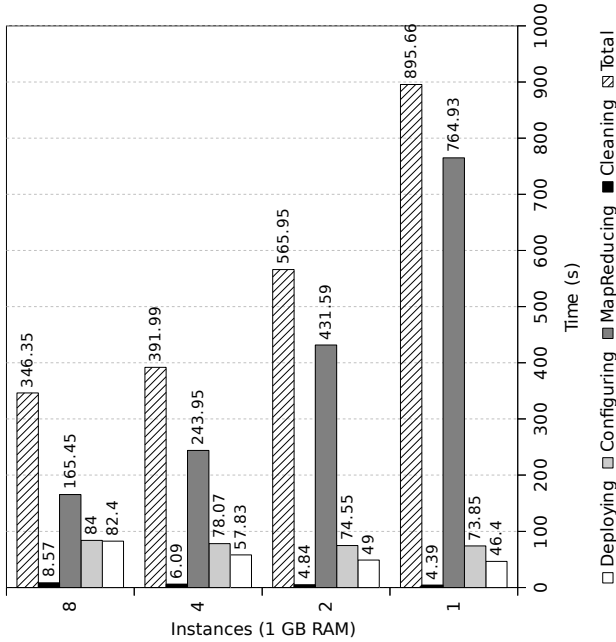
Fig. 6. Tendency on scaling out

In figure 7 it is shown how vertical scaling affects qosh performance. Deploying and cleaning times stay constant (two instances on every execution), whereas processing and configuring times are reduced as instances are upgraded.

Looking at both charts it can be seen that, for our particular dual-node deployment, vertical scaling uses infrastructure resources more efficiently: two 4 GB RAM/VCPU instances are
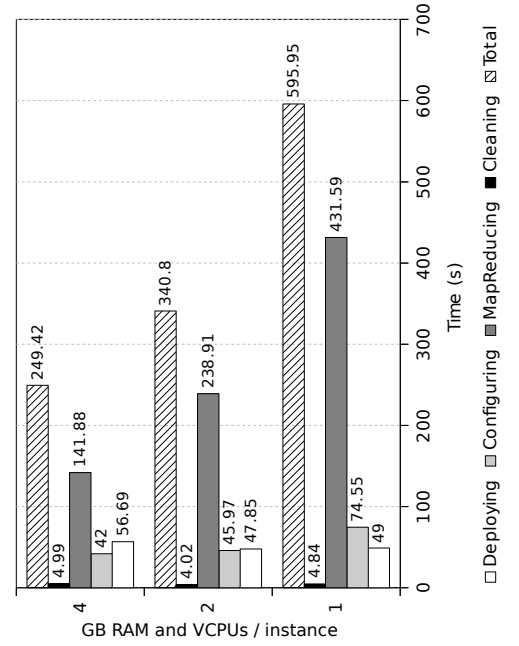
Fig. 7. Tendency on scaling in

roughly equal to eight 1 GB RAM/VCPU, but total executing time is reduced by almost 28%.

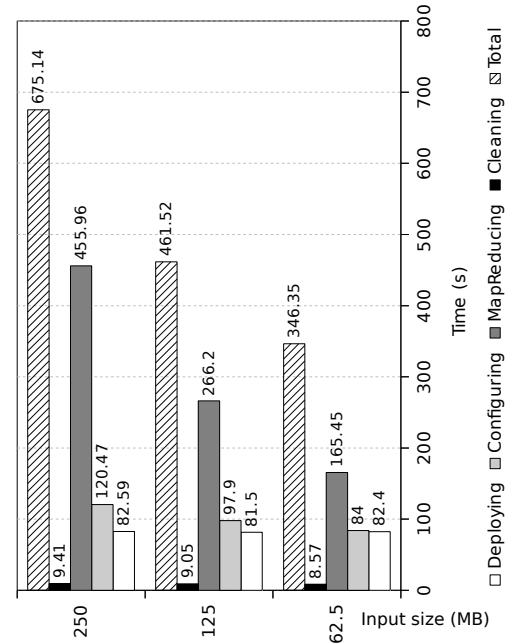Finally, figure 8 shows timing evolution as input size heightens.

Fig. 8. Tendency on input size escalation

V. CONCLUSIONS AND FUTURE WORK

qosh's main contributions to the current *cloudy* landscape might be summarized in the following points.

- Simplicity of installation and exploitation. The qosh installation script cannot be more straightforward and its web interface provides the means to define custom virtual Hadoop deployments, to launch MapReduce jobs and to download results easily and intuitively.
- Vertical integration of every module, from the web interface to file storage and infrastructure provisioning. The qosh installation script configures automatically a complete execution environment with no user intervention required.
- High performance on initial setup. qosh has been conceived from the beginning to execute MapReduce workflows on virtual clusters, while maintaining a minimum learning curve. Other similar solutions that ease the complexity to install and configure cloudy environments like *DevStack* [12], do not provide an usable environment to develop applications on top of an IaaS cloud.
- Reusability of the main components. Adhering to *Amazon Web Services* standards when building the VM [11] allows to deploy it over any EC2-compatible cloud IaaS. Aditionally, its worth considering the ability to make the VM run with no cloud architecture underneath, by setting up an installation on top of a real cluster.
- Adaptability to handle infrastructure provided by other cloud frameworks, and thus, opening the possibility to spread virtual clusters on hybrid clouds just by rewriting the *Compute* module.
- Transparency in every way possible. Source code and documentation both from qosh and its required libraries and applications, are publicly available online [13].

A. Future work

Arguably, the Achilles heel of qosh is a certain coupling between modules. It would be interesting to abstract a REST client interface in order to code different *delegates*, which could be used to adapt messages to any REST API language, clearing hybrid cloud implementations up.

Following the same decoupling dynamics, some use cases like view processing in *Django* could be detached from the web interface as action objects, and be executed in an action processor. This action processor could be shaped, for instance, a processes pool consuming these action objects queuing in memory easing off horizontal scaling and load balancing.

REFERENCES

[1] Jeffrey Dean and Sanjay 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.
[2] Amazon Web Services: Elastic MapReduce. http://aws.amazon.com/elasticmapreduce/, October 2013.
[3] Pierre Riteau, Ancuta Iordache, and Christine Morin. Resilin: Elastic MapReduce for Private and Community Clouds. Research Report RR-7767, INRIA, October 2011.
[4] OpenStack: Project Savanna. https://wiki.openstack.org/wiki/Savanna, October 2013.
[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] 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.
[7] OpenStack. http://www.openstack.org/, October 2013.
[8] Apache Cloudstack. http://cloudstack.apache.org/, October 2013.
[9] R. Moreno-Vozmediano, R. S. Montero, and I. M. Llorente. IaaS Cloud architecture: From virtualized datacenters to federated Cloud infrastructures. *IEEE Computer*, 45:65–72, December 2012.
[10] Daniel Nurmi, Rich Wolski, Chris Grzegorczyk, Graziano Obertelli, Lamia Youseff, and Dmitri Zagorodnov. The Eucalyptus open-source Cloud-computing system. In *9th IEEE International Symposium on Cluster Computing and the Grid*, September 2009.
[11] Apache Hadoop 1.0.4 based on CentOS 6.3 VM. https://drive.google.com/file/d/0B2lmVzXW-C5UcmZIYk80dTZJb0k/edit?usp=sharing, October 2013.
[12] DevStack. http://devstack.org/, October 2013.
[13] qosh main page. https://code.google.com/p/quick-openstacked-hadoop/, October 2013.