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
Brookhaven Laboratories
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 [3] 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.

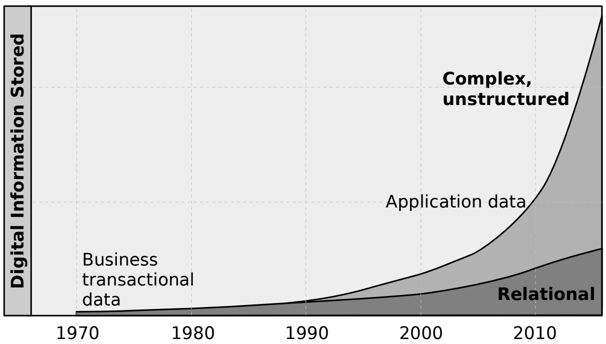

Figure 1: Unstructured and relational data volume evolution. Source: *Cloudera Inc.*

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 non-

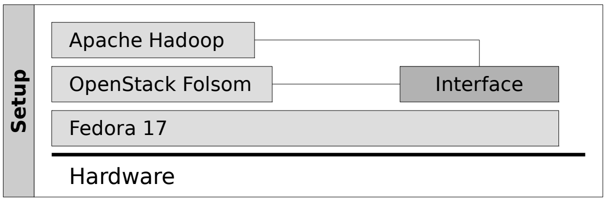

Figure 2: High level design diagram

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 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* [6], *Savanna* [2] or *Dynamic MapReduce* [5], or by reusing its cloud interface to build a *MapReduce* platform upon like with *Cloud MapReduce* [4].

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 details *qosh*'s architecture and self-installing deployment structure.

2. ARCHITECTURE

qosh's setup defaults to a single node installation in which both infrastructure and execution environment is 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.

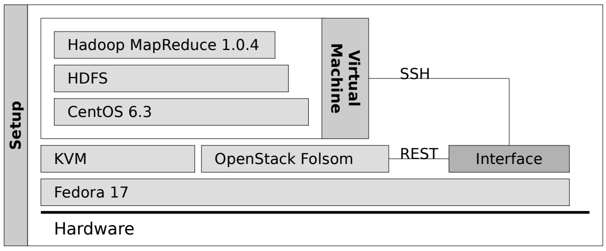

Figure 3: Layered initial deployment

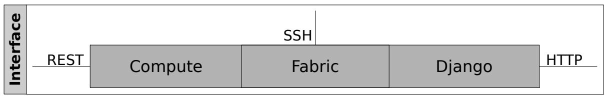

Figure 4: Interface composition

2.1 Initial deployment

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.

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

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

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.

2.2 Interface

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

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 would be allowed to manually define the deployment of VMs.

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.

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

3. APACHE HADOOP VIRTUAL MACHINE

The *Apache Hadoop* installation has been manually configured from scratch inside a virtual machine, in such a way that it could be run on top of any *Amazon Elastic Compute Cloud* (EC2) compatible *IaaS* service, with no theoretical limit to horizontal or vertical scaling. What follows is the procedure carried out to yield the VM image.

- After a clean Fedora 17 installation, `yum` was employed to add the *Virtual Machine Manager* (`virt-manager`) package which would be exerted to sketch the VM. Along with it, `libvirt`, `kvm` and `qemu` were also installed.
- Using `virt-manager` a VM was spawned anew with 1 VCPU, 1 GB RAM and 4 GB qcow2 HDD.
- A *CentOS 6.3* network installation image was booted inside the VM, choosing *Basic Server* as set of packages to be configured within a single *ext4*-formatted partition without LVM.
- Once completed and self-restarted, the system was updated and *SELinux* relaxed to be *permissive* by issuing:

```
[guest]$ sudo yum update -y
[guest]$ sudo setenforce 0
[guest]$ sudo sed -i \
s_SELINUX=enforcing_SELINUX=permissive_ \
/etc/selinux/config
```

- Both *Oracle JRE 1.7* and *Apache Hadoop 1.0.4* were downloaded —AMD64 and rpm packages—, installed and configured in the VM.
- Right afterwards, an `hduser` was added to `hadoop` as primary group by typing:

```
[guest]$ sudo useradd -g hadoop hduser
[guest]$ sudo passwd hduser
```

- `sshd` was set to disallow either `root` user connections or the tuple (user, password) as credentials, to effectively limit SSH tunnels to those authorized with (public, private) keypair collations. By using this approach, only the user spawning a virtual cluster would have access to its instances, provided he or she remained the sole acquaintances with the *OpenStack*-injected private keypair.

```
[guest]$ sudo sed -i \
s_^#PermitRootLogin\ yes.*_\
PermitRootLogin\ no_ /etc/ssh/sshd_config
[guest]$ sudo sed -i \
s_^PasswordAuthentication\ yes.*_\
PasswordAuthentication\ no_ \
/etc/ssh/sshd_config
[guest]$ sudo rm -rf /home/hduser/.ssh
[guest]$ sudo rm -f /etc/ssh_host*
```

- Three scripts were written (refer to appendix A) and placed in `/etc/init.d/` to convey user configuration to the VM and to secure access.
- With `yum groups`, unused sets of services and applications —like `Xserver`— were removed from the system in order to trim its size and memory footprint.
- Subsequently, the qcow2 HDD image was packed in two steps. Initially, the beginning offset of the partition inside the image was trimmed by exposing only that partition with `qemu-nbd`. Right afterwards, its contents were dumped into a new partitionless image. Then, a long zero-file was generated to fill up all the remaining free space in the image, so that `qemu-nbd` be adequately executed to do the real compressing. Once `qemu-img` had completed processing, the image file in the host system was neatly compressed but the image's file system reported no free space, clogged up with the zero-file that had to be manually removed.

```
[host]$ sudo modprobe nbd max_part=8
[host]$ sudo qemu-nbd -c /dev/nbd0 \
-P 1 original.qcow2
[host]$ dd if=/dev/nbd0 of=trimmed.qcow2
[host]$ sudo qemu-nbd -d /dev/nbd0

[host]$ sudo qemu-nbd -c /dev/nbd0 \
trimmed.qcow2
[host]$ sudo mkdir /mnt/img
[host]$ sudo mount /dev/nbd0 /mnt/img
[host]$ sudo dd if=/dev/zero \
of=/mnt/img/root/zeros bs=1M count=4K
[host]$ sudo umount /mnt/img
[host]$ sudo qemu-nbd -d /dev/nbd0
[host]$ qemu-img convert -c -p -f qcow2 \
-O qcow2 trimmed.qcow2 compacted.qcow2

[host]$ sudo qemu-nbd -c /dev/nbd0 \
compacted.qcow2
[host]$ sudo mount /dev/nbd0 /mnt/img
[host]$ sudo rm -f /mnt/img/root/zeros
```

- Lastly, both *initram* and *kernel* were copied out to the host machine and `nbd0` disconnected.

```
[host]$ sudo cp /mnt/img/boot/initramfs-\
$(uname -r).img
/mnt/img/boot/vmlinuz-$(uname -r)
[host]$ sudo umount /mnt/img
[host]$ sudo qemu-nbd -d /dev/nbd0
```

4. EXECUTION

5. PERFORMANCE

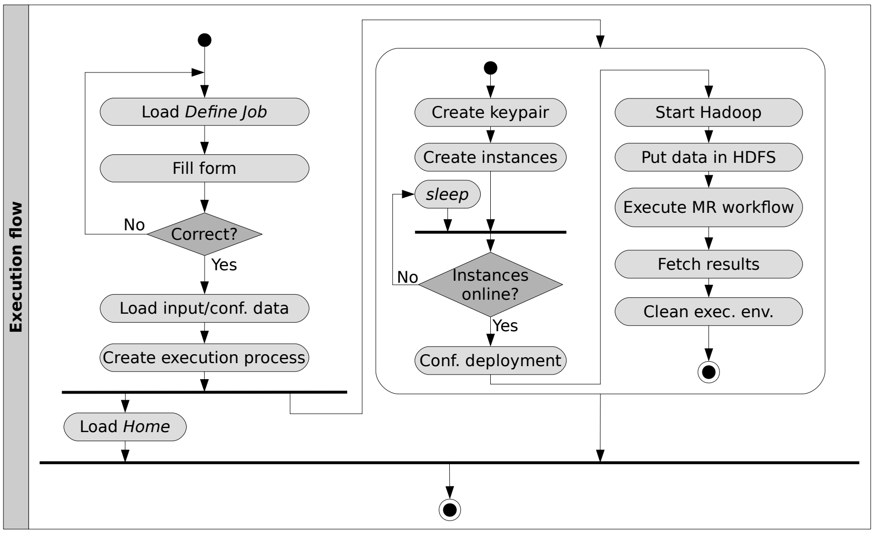

Figure 5: Global execution flow

6. REFERENCES

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

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

[3] 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.

[4] H. Liu and D. 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] 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.

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

APPENDIX

A. CLOUD-INIT SCRIPTS

A.1 cloud-prenet.sh

```
#!/bin/sh
#
# Custom script to control pre-network
# initialization
#
# Remove ssh Keys DSA, RSA & HOST
rm -f /etc/ssh/*host*

# Remove persistent-net-rules
rm -f /etc/udev/rules.d/70-persistent-net.rules

# Remove history
rm -f /home/hduser/.bash_history\
 /root/.bash_history
```

A.2 cloud-init.sh

```
#!/bin/sh
#
# Simple cloud-init script to fetch
# meta-data injected into instances
# in order to configure them

fetch_and_set() {

  response_code="$(\
   curl -sI http://169.254.169.254/1.0/\
   meta-data/hostname | grep HTTP\
   | awk {'print $2'})"

  if [ "$response_code" == "200" ]
  then
    new_hostname="$(\
     curl -s http://169.254.169.254/1.0/\
     meta-data/hostname)"
    sed -i 's/'"$HOSTNAME"'/'"$new_hostname"\
     '/g' /etc/hosts /etc/sysconfig/network
    HOSTNAME=$new_hostname
    hostname $new_hostname
  fi

  response_code="$(\
   curl -sI http://169.254.169.254/1.0/\
   meta-data/public-keys/0/openssh-key\
   | grep HTTP | awk {'print $2'})"

  if [ "$response_code" == "200" ]
  then
```

```
    rm -rf /home/hduser/.ssh
    mkdir -p /home/hduser/.ssh

    curl -s http://169.254.169.254/1.0/\
     meta-data/public-keys/0/openssh-key\
     | grep 'ssh-rsa' >> \
     /home/hduser/.ssh/authorized_keys

    chown -R hduser:hadoop /home/hduser/.ssh
    echo -e "\nAUTHORIZED_KEYS:"
    echo "************************"
    cat /home/hduser/.ssh/authorized_keys
    echo "************************"
  fi

}

# Change hostname to match meta-data
# instance name if found
#
http_response_code="$(\
 curl -sI http://169.254.169.254/1.0/\
 | grep HTTP | awk {'print $2'})"

[ "$http_response_code" != "200" ] && \
echo -e "\nWARNING: No meta-data found!\n"\
 && exit 1

fetch_and_set
```

A.3 cloud-shtdwn.sh

```
#!/bin/sh
#
# Remove ssh Keys DSA, RSA & HOST
rm -f /etc/ssh/*host*

# Remove ssh authorized_keys
rm -f /root/.ssh/authorized_keys\
 /home/hduser/.ssh/authorized_keys

# Remove persistent-net-rules
rm -f /etc/udev/rules.d/\
70-persistent-net.rules

# Remove history
rm -f /home/hduser/.bash_history\
 /root/.bash_history

# Remove hadoop logs
rm -rf /var/log/hadoop/hduser

# Clear some system logs
for file in $(ls -F /var/log |grep -ve "/$")
do
  echo > /var/log/$file
done
```