A Self-managed Mesos Cluster for Data Analytics with QoS Guarantees

Sergio López-Huguet[a,*], Alfonso Pérez[a], Amanda Calatrava[a], Carlos de Alfonso[a], Miguel Caballer[a], Germán Moltó[a], Ignacio Blanquer[a]

[a]*Instituto de Instrumentación para Imagen Molecular (I3M)*
Centro mixto CSIC - Universitat Politècnica de València
Camino de Vera s/n, 46022, Valencia

Abstract

This article describes the development of an automated configuration of a software platform for Data Analytics that supports horizontal and vertical elasticity to guarantee meeting a specific deadline. It specifies all the components, software dependencies and configurations required to build up the cluster, and analyses the deployment times of different instances, as well as the horizontal and vertical elasticity. The approach followed builds up self-managed hybrid clusters that can deal with different workloads and network requirements. The article describes the structure of the recipes, points out to public repositories where the code is available and discusses the limitations of the approach as well as the results of several experiments.

Keywords: Cloud Orchestration, Elasticity, Quality of Service, Data Analytics, Hybrid Clusters

1. Introduction

The need for data analytics platforms has risen in the recent years, in parallel to the increase in the computing and data storage requirements, in order to tackle the challenges of data processing. Configuring and operating such platforms is not straightforward and requires non-trivial system administration skills. Data analytics platforms involve multiple components and resources, which must be appropriately linked and cross-configured. In addition, dealing with unpredictable workloads is an operationally complex task that requires dynamically readjusting the resources and reconfiguring them on the fly.

In this way, this article presents a set of tools and configuration recipes for deploying a virtual self-managed cluster of computing nodes. The cluster can scale horizontally (in and out), by adding and removing computing resources and reconfiguring them according to the workload, and vertically (up and down), by readjusting the assigned resources to individual jobs dynamically to satisfy a given Quality of Service (QoS).

*Corresponding author: Tel. +34963877356

Email addresses: `serlohu@upv.es` (Sergio López-Huguet), `alpegon3@upv.es` (Alfonso Pérez), `amcaar@i3m.upv.es` (Amanda Calatrava), `calfonso@upv.es` (Carlos de Alfonso), `micafer1@upv.es` (Miguel Caballer), `gmolto@dsic.upv.es` (Germán Moltó), `iblanque@dsic.upv.es` (Ignacio Blanquer)

This paper introduces the problem, the software architecture, the automatic deployment tools and recipes, the elasticity mechanism and the experiments, discussing the results obtained. The reminder of the paper is structured as follows. First, section 2 examines the requirements of a data analytics platform and revises the state of the art related to the work presented in the paper. Then, section 3 presents the proposed architecture of the platform used to perform data analytics and the mechanisms involved in the elasticity management. Also, a brief analysis of each component involved in the architecture is presented in this section. Section 4 describes the most relevant metrics obtained from the deployment of the self-managed virtual cluster and the execution of several test cases to validate the horizontal and vertical elasticity. Section 5 discusses the main developments and improvements presented in this work in comparison with the state of the art. Finally, section 6 summarizes the main results, concludes the paper and points to future work.

2. Requirements & State of the art

This section presents the requirements and reviews the state of the art of the two main areas of research that constitute the basis of this work (cloud orchestration and elastic clusters) as well as other cloud-based processing software architectures that address the requirements identified.

2.1. Requirements

In this work, we consider three types of use cases that address three main problems in data analytics [1]. The first use case is data acquisition, which deals with the periodic acquisition of external datasets and the integration with the previously acquired data. The second use case is the development of descriptive models, aiming at deriving additional information and knowledge from raw data. Finally, the third use case concerns predictive models, which build up models for estimating specific variables under new scenarios.

From this analysis, we identified the following technical requirements concerning processing:

1. Running unrestricted batch jobs. This requirement refers to the execution of batch jobs that do not have any QoS guarantee to meet, such as long-running jobs that are not linked to a production service.
2. Running periodic batch jobs. Periodic workloads, such as daily jobs that retrieve the updated data from a public data source, must be regularly executed by the platform.
3. Running batch jobs with QoS restrictions. Tracking the job progress is a complex task that is limited to jobs that use a specific execution framework that supports it (e.g. Spark, Marathon, etc.). We consider in this requirement the guarantee that a given amount of CPU time is assigned to a running job in a given time frame.
4. Self-adapting elasticity. This requirement is strongly linked to requirement 3. The platform should provide enough resources to deal with new jobs and to ensure that jobs with QoS are properly executed.

5. Running parallel Spark jobs. The platform must support the execution of Spark jobs across several nodes in parallel, providing the right amount of resources to each job.

Considering these requirements, we focus on analyzing the available technologies for cloud orchestration, elasticity and cloud services for data analytics.

2.2. Cloud Orchestration

Cloud orchestration is the process needed to automate the entire lifecycle of a cloud application. It implies the deployment of all the computational resources, the installation and configuration of the different component parts of the application and their correct interconnection.

To describe cloud applications, the Topology and Orchestration Specification for Cloud Applications (TOSCA) [2] open standard has been defined by the OASIS consortium[1]. It defines the interoperable description of services and applications to be run on the cloud, including their components, relationships, dependencies, requirements, and capabilities; thereby enabling portability and automated management across cloud providers regardless of the underlying platform or infrastructure.

By using TOSCA to model the user's complex application architectures it is possible to obtain repeatable and deterministic deployments. Users can port their virtual infrastructures among cloud providers obtaining the same expected topology.

Several open source orchestration tools and services exist in the market, but most of them come with the limitation of only supporting their own Cloud Management Platforms (CMPs) because they are developed within those project ecosystems. As an example we can cite some of them, such as OpenStack Heat [3] and its YAML-based Domain Specific Language (DSL) called Heat Orchestration Template (HOT) [4], native to OpenStack [5]. OpenNebula [6] also provides its own JSON-based multi-tier Cloud application orchestration called OneFlow [7]. Eucalyptus [8] supports orchestration via its implementation of the Amazon Web Services (AWS) CloudFormation [9] web service.

In case of other general orchestration tools, we can find Cloudify [10], which provides TOSCA-based orchestration across different cloud providers. Unfortunately Cloudify is not currently able to deploy on OpenNebula sites, one of the main CMPs used within current science clouds. Apache ARIA [11] is a very recent project, not mature enough and also without support for OpenNebula. Project CELAR [12] uses an old XML-based TOSCA version with SlipStream [13] as the orchestration layer (this project has no activity in the last years and SlipStream has the limitation of being open-core, thus not supporting commercial providers in the open-source version). CompatibleOne [14] provided orchestration capabilities based on the Open Cloud Computing Interface (OCCI). However, the project has not been active in the last years. OpenTOSCA[15] currently only supports OpenStack and the AWS EC2 [16] service.

Our previous work in the field is the Infrastructure Manager (IM) [17], a cloud orchestration runtime that deploys complex and customized virtual infrastructures on multiple back-ends. It supports the TOSCA Simple Profile in YAML version of the standard. It is compatible with a wide variety of Cloud back-ends, both on-premises CMP and public

[1]https://www.oasis-open.org/

Cloud providers, thus making user applications cloud agnostic. In addition, it features DevOps capabilities, based on Ansible[2] to enable the installation and configuration of all the user required applications providing the user with a fully functional infrastructure.

2.3. Cloud-based Data analytics

In recent years, the Data Deluge [18] made it possible to enter an era in which distributed computing is now the new normal, paving the way for Big Data, a term coined for scenarios in which the amount of data (or the speed at which data is generated) can no longer be processed in a feasible time in a single computer. Google, being a large-scale data-oriented enterprise, faced the challenges that involved the processing of huge datasets and, in 2008 unveiled the MapReduce programming model [19], together with an associated implementation for processing large datasets. This was the seed that made possible the Apache Hadoop project [20], an open-source software for reliable, scalable, distributed computing that ended up forming the kernel (as is the case of the Hadoop Distributed File System) of a huge ecosystem of tools aimed at solving Big Data problems. This is the case of Hive [21], a data warehouse software for querying and managing large datasets in a distributed storage or Pig [22], a platform for analyzing large datasets via a high-level language for expressing data analysis programs. Other platforms such as Spark [23], due to its speed related to in-memory processing, are also fundamental for many Big Data scenarios.

In addition, the trend towards lightweight virtualization allowed container technology to considerably evolve, exemplified by the recent advances in Linux containers (LXC) [24] and Docker [25], and the HPC-specific Singularity [26]. LXC enables to run multiple isolated processes in one host without the overhead caused by the hypervisor layer introduced by Virtual machines (VMs) in CPU, memory and storage [27], as if it was a whole new machine. Docker is oriented to applications, and the underlying idea is to run a single application that is isolated and with a tailored environment. Moreover, the ecosystem of tools around Docker has exploded in the last years, with contributions in many areas such as Continuous Integration/Continuous Delivery (CI/CD), application packaging and container orchestration tools. Indeed, there are many applications to manage the execution of containers across multiple hosts (e.g. Kubernetes [28] or Deis [29]) but one of the most advanced tools for computationally challenging problems is Apache Mesos [30], a software that abstracts CPU, memory, storage and other compute resources away from machines to enable fault-tolerant distributed systems to be built. Moreover, Mesos supports several frameworks suitable for resource-intensive computing, as is the case of Chronos [31], for the job fault-tolerant executions, and Marathon [32], for the execution of long-running services. Finally, Singularity is an alternative to Docker that has been developed in the HPC context. Its growing popularity is due to the ability to create containers that run in the user space, and are integrated with the underlying system by mapping the system user ids and important folders (such as home). In order to foster its usage, it is able to get images in Docker format, among others.

2.4. Elastic Clusters

Elasticity is the property of an infrastructure to dynamically adapt itself to the current or estimated workload. This is manifested in cloud infrastructures at several levels. In the

[2]https://www.ansible.com/

lower level of on-premises clouds, elasticity represents the ability to dynamically power on and off the nodes of the underlying hardware in order to provision and relinquish physical computing hardware on which the virtualized infrastructure will run. At the level of IaaS (Infrastructure as a Service), these techniques should be integrated within the Cloud Management Frameworks (CMF) so that requests of virtual infrastructure deployment trigger, if necessary, the powering of physical nodes in order to accommodate the virtual infrastructure that will be executed on top of the physical infrastructure. Horizontal elasticity is the ability to dynamically deploy and terminate nodes within a virtual infrastructure according to a set of elasticity rules (scale in/scale out) and this is exemplified by services such as Auto-Scaling [33] for AWS or Heat/AutoScaling for OpenStack, to name a few.

In the literature, we can find several research works regarding horizontal elasticity in virtual clusters. In [34] and [35], the Nimbus toolkit is employed to implement a tool to create elastic sites, so that physical clusters based on a Local Resource Management System (LRMS) such as Torque are supplemented with computational resources provisioned from AWS according to different policies.

A widely used tool is StarCluster [36] which enables the creation of virtual clusters in ÅWS, that satisfy a user-defined list of required applications (Sun Grid Engine, OpenMPI, NFS, etc.). The Virtual Machines (VMs) are based on predefined Amazon Machine Images (AMI). In addition, a plugin named Elastic Load Balancer [37] is available to add and terminate new cluster nodes taking into account the number of jobs queued up at the LRMS. The main limitation of this plugin is that it requires a permanent connection to the cloud infrastructure from the StarCluster installation in the user's computer in order to deploy and terminate the VMs.

In the last years, horizontal elasticity has also been introduced in well-known Big Data frameworks. This is the case presented in [38], where the authors propose a system called BBQ, which is able to provide elasticity to Hadoop MapReduce. It works with AWS and needs a specific modified implementation of Hadoop to properly work with BBQ, thus, limiting the ability to choose a desired configuration for the users.

Our previous work in the field is Elastic Cloud Computing Cluster (EC3) [39], a tool that creates elastic virtual clusters from computational resources provisioned from IaaS clouds. These clusters scale out to a larger number of nodes on demand, up to a maximum size specified by the user. Whenever idle resources are detected, the clusters dynamically and automatically scale in, according to some simple policies, in order to minimise the costs in the case of using a public cloud provider.

To dynamically manage the clusters, EC3 relies on the CLUster Energy Saving (CLUES)[40] tool, an elasticity manager. CLUES has been already integrated in public and on-premises cloud environments in order to deploy/destroy VMs and it is able to automatically integrate the VMs in the LRMS according to the workload of the cluster.

Horizontal elasticity is appropriate when the problems solved are inherently parallel. In cases where the problems cannot benefit from an increase in the amount of resources, another elasticity strategy must be considered. Vertical elasticity is the ability to dynamically resize the resources of the nodes, such as the number of CPUs, the share of CPU or memory, according to a set of elasticity rules (scale up or scale down).

Most of the hypervisors and cloud IaaS support vertical elasticity. This is the case of OpenNebula and OpenStack, which offer functions to resize the memory or change the number of CPUs of stopped VMs. Nevertheless, dynamic resizing of VMs is not supported because it is necessary to act both at the level of the hypervisor and at the level of the VM's operating system.

There are techniques for providing vertical elasticity leveraging the CPU CAP (the maximum amount of CPU resources a VM can use) and the physical memory allocated by the hypervisor. This strategy only acts at the level of the hypervisor and, thus, it has a better approach than the strategy described above. This way, the internal configuration of the VM remains the same, but it is provided with a higher share of physical resources, so the virtual CPU can run faster or slower, and have more RAM mapped on physical RAM. For example, the work by Shen et al. [41] describes an approach named CloudScale to automate fine-grained elastic resource scaling for multi-tenant cloud computing infrastructures. Other examples of these techniques are described in [42]. It should be pointed that, in this example, the system needs access to the private network to connect with the worker nodes and root privileges for leveraging the CPU CAP and the memory RAM.

Vertical memory elasticity is interesting for some problems (e.g. when the consumed memory grows over time). For this purpose, the virtualization hypervisors provide two mechanisms: add or remove memory, also named hot memory plugging, and memory ballooning ([43], [44]). In any case, the allocation of resources with the aforementioned techniques affects all of the tasks running in the VM.

Providing vertical elasticity to guarantee QoS restrictions requires a more fine-grained approach than the techniques described above because it is necessary to resize the assigned resource for a specific job (not all the running jobs of the VM). For this purpose, it is needed to act at the level of processes of the operating system of the VM.

Nowadays, Docker containers are becoming the new platform for packaging, distribution and deployment of applications in Cloud Computing. Vertical elasticity in Docker containers has few works in the literature compared those available for VMs. The work by Al-Dhuraibi et al. [45] presents a mechanism that modifies the allocated resources (CPU time, vCPU cores and memory) of a Docker container according to the workload demand. This mechanism monitors the CPU time, CPU utilization, vCPUs and memory utilization to take the elasticity decisions and implements these decisions modifying the *cgroups* pseudofiles of the Docker container directly.

The job execution capabilities of the Mesos cluster are provided by their frameworks and, commonly, these jobs are encapsulated in Docker containers or Mesos native containers, which are processes of the VM. In case of Mesos, the level of processes of the operating system of the VM can be seen as the jobs of frameworks that support vertical elasticity (for example, Marathon).

Some techniques modify the assigned resources for Apache Mesos execution frameworks. One example of this approach is [46], a Mesos executing and monitoring framework called Makeflow is designed to adjust the number of vCPUs of a series of independent jobs according to their actual performance. The work [47] proposes a mechanism to provide both horizontal and vertical elasticity according to the share of CPU and memory used. This technique considers that the jobs do not store a persistent state and, thus, they can be easily restarted.

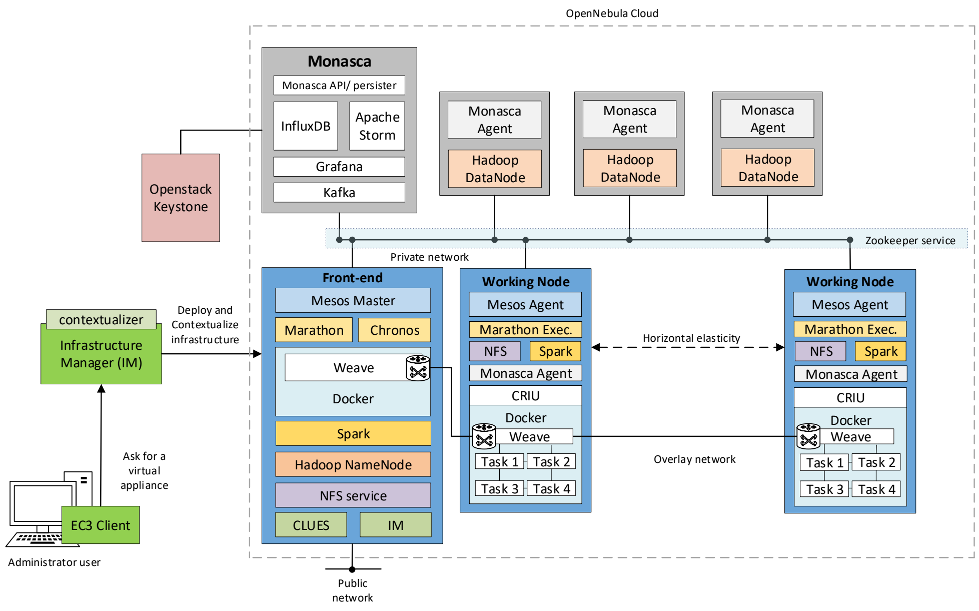

Figure 1: Architecture of the required infrastructure to perform data analytics.

3. Architecture Design

The architecture of the infrastructure must enable the execution of a wide variety of workloads, that range from parallel to high-throughput jobs, including short jobs and big data workflows. This section provides information on the proposed architecture for the deployment and the automatic management of both horizontal and vertical elasticity at the level of the framework.

3.1. General Architecture

The proposed architecture is depicted in Figure 1. In the scenario, the administrator user is in charge of deploying the virtual infrastructure by using the EC3 client. EC3 interacts with the Infrastructure Manager (IM), requesting the needed resources considering the characteristics of the cluster together with its specific configuration. With these data, the IM interacts with the selected cloud provider, requests the VMs that compose the cluster and configures them. Notice that, by using IM, the cluster can be deployed on different on-premises and public clouds. In particular, at least the following providers can be used: OpenNebula, OpenStack, Amazon Web Services, Google Cloud Platform [48] and Microsoft Azure [49].

The infrastructure has three main types of nodes:

- The front-end node, that is the master of the cluster. It contains the Mesos master instance together with the Marathon and Chronos frameworks, and also including Docker, Hadoop, Spark and NFS. The front-end also has an instance of the IM that, together with CLUES, is in charge of managing the elasticity of the cluster. This node is also in charge of offering an interface for end users of the infrastructure.

- The Monasca [50] nodes, that include the Openstack Monasca server instance, that also have Apache Kafka [51], Apache Storm [52], InfluxDB [53] and Grafana [54]; three Monasca agents also act as Hadoop datanodes. The architecture also includes a VM that provides the keystone [55] server needed by Monasca. It is not running inside the Monasca server to avoid excessive resource consumption.
- The working nodes, that are the elastic part of the infrastructure. These nodes are deployed on-demand when triggered by CLUES, that monitors the Mesos and Marathon queues and reactively provides the needed Mesos agents. These working nodes also contain Docker and CRIU [56], to have the ability to run jobs inside containers with checkpointing capabilities. NFS is employed to provide a shared file system across the nodes to manage the checkpointed containers. A Spark daemon is also running on each working node and all of them are monitored by Monasca.

Regarding networking, all the components of the infrastructure are interconnected by a private network. Moreover, a dedicated overlay network, managed by Weave [57], is created to interconnect the containers running on different hosts. The front-end is also connected to this overlay network, so that applications running in the containers can interact with the services using the overlay network. This guarantees a bi-directional communication on a wide range of ports without exposing the jobs to the Internet. The front-end of the cluster is the only component that can be accessed via a public network, even though the whole infrastructure has access to the Internet via NAT (Network Address Translation).

3.2. Vertical scaling

This section describes the architecture of the developed mechanism for providing vertical elasticity to the batch jobs with QoS restriction. As it is described in section 2, the developed mechanism aims to guarantee that the desired amount of CPU time is assigned to a running job in the given time frame of the targeted QoS. These jobs are embedded in Docker containers and are executed using the Marathon framework of Mesos. The architecture is composed of three main components: Launcher, Executor and Supervisor. This architecture is depicted in Figure 2, where the green dashed lines represent the interactions between components and the other services (Marathon, Monasca and Keystone).

Vertical elasticity is the ability to resize the assigned resources of a job in order to meet a targeted QoS. In this work, the mechanism varies the assigned share of CPU to the job. Resizing jobs in Marathon requires updating the job specification in the Marathon scheduler via its REST API. Once the job specification is changed, Marathon removes the older version of the job without preserving the execution state. Then, it runs the job with the new resource reservation. Furthermore, it should be pointed out that the new job execution can run on another working node.

Therefore, everytime a job is resized its progress state is lost. To avoid this problem, this work uses CRIU [56], which is a project for the Linux operating system that allows to freeze a running application as a collection of files called checkpoint. Checkpointing allows users to stop and resume the job at the same execution point as it was when the checkpoint was made, even in another machine. As Marathon cannot freeze and resume the Docker container using CRIU, it is required that the developed mechanism manages the Docker container execution.

The Launcher is a command-line tool in charge of the submission of the job. Users run the Launcher specifying the job in JSON format with the QoS information, the parameters to connect and configure the Supervisor, and the credentials of the Marathon scheduler. The QoS information is composed of the number of seconds of CPU time that the mechanism should assign for completing the job, the over-progress percentage, and the time frame (in seconds) for executing the job. Users can configure, with the parameter called over-progress percentage, the *overprogress* threshold used by the Supervisor in Algorithm 1 for computing the job *performance state* .

First, the Launcher assigns to each job a unique identifier (UUID). If the Launcher submits to Marathon the job specification provided by the user, then the Marathon executor will manage the Docker container. As the mechanism must manage the Docker container for using the checkpointing feature, an additional component, the Executor, is required. To allow the Executor to manage the Docker container, the Launcher creates a new job specification based on the job specification provided by the user. Tasks in Mesos are isolated because they are executed embedded into Docker containers or Mesos native containers. Thus, the Marathon executor runs the new job (the Executor) created by the Launcher isolated by a Mesos native container. Once the new job specification is generated, the Launcher submits it via a REST API call to the Marathon scheduler. Finally, the Launcher sends a message with information about the job to the Supervisor. This information is formed by the job name, the job UUID, the maximum overprogress percentage, the number of seconds of CPU time that the mechanism should allocate for completing the job, and the time frame for executing the job in seconds.

The Executor performs several tasks. First, it prepares the worker node to enable the Docker container monitoring using a modified Docker plug-in of the Monasca Agent. Afterwards, it checks for the existence of a previous checkpoint of the job in the directory shared by all of the worker nodes. If the Executor does not find a checkpoint, then it starts the Docker container. Once the Docker container is running, the Executor notifies via a REST API call to the Supervisor. Then, it waits until the Docker execution is done or until capture the termination signal sent by the Marathon executor when a scaling decision is implemented by the Supervisor. If the Docker container ends its execution, the Executor cleans the worker node and the shared directory, and notifies the end of the job execution via a REST call to the Supervisor. If the Executor captures a termination signal, then it means that the Docker container will be resized. Thus, the Executor performs the checkpoint of the execution and stores it into the directory shared by all the worker nodes. Once the Marathon executor runs the Executor with the new allocated resources, the Executor resume the Docker container execution from the stored checkpoint.

The Supervisor is a REST service in charge of the decision making. When the Executor notifies the start of the job execution to the Supervisor, it begins to periodically monitor the job to decide if scaling up or down is needed. There are three possible job performance states: *overprogress*, *underprogress* or *ontime*. If the job state is *overprogress* or *underprogress*, then the Supervisor scales, respectively, down or up the assigned resources to the job. The Supervisor implements the scaling decision re-submitting the job specification (which is available on the Marathon scheduler) with the new resource reservation by a REST API call to the Marathon scheduler. The amount of share of CPU that is incremented or decremented is set at the startup of the Supervisor. For this work, empirical observation indicates that 0.4 offers good results. Once the new resource assignation arrives to the Marathon scheduler, it notifies the Marathon executor to send the termination signal to the old version of the job, which is captured by the Executor to create and store the checkpoint.

The Supervisor uses the Algorithm 1 to determine the job *performance state*. This algorithm has three input values: the *performance*, the *overprogress*, and *underprogress* thresholds. The *performance* is obtained using the Equation 2. The Equation 2 uses the CPU time consumed $cputime_{current}(t)$ and the expected CPU time consumed $(cputime_{desired}(t))$ on a certain time t (both expressed in seconds). The Supervisor estimates the CPU time consumed at certain time t, $cputime_{current}(t)$, requesting via the REST API the gathered information about the Docker container to the Monasca. The information obtained by the Supervisor from Monasca is composed of two metrics (*container.cpu.user_time* and *container.cpu.system_time*). These values correspond to the total user and system clock ticks consumed by the container in the node where it is running. These values are transformed into seconds, dividing them by the clock ticks per second constant of the system. The addition between these values is the $cputime_{current}(t)$. In addition, the Supervisor also estimates the CPU time that the job would have to consume at certain time t, $cputime_{desired}(t)$, by means of Equation 1.

$$cputime_{desired}(t) = \begin{cases} \frac{(t_{current} - t_{start}) * seconds_{job}}{seconds_{timeframe}} & \text{if } t_{current} \leq t_{start} + seconds_{timeframe} \\ seconds_{job} & \text{otherwise} \end{cases} \tag{1}$$

where $t_{current}$ is the current time in timestamp format, t_{start} is the start time of the execution in timestamp format (obtained when the Executors notifies the start of the execution), $seconds_{job}$ is the number of seconds of CPU time that the mechanism should allocate for completing the job (sent by the Launcher), and $seconds_{timeframe}$ is the available time interval to complete the execution of the job (sent by the Launcher).

The *underprogress* threshold is 10% by default, so the value used in Algorithm 1 is 0.9. The *overprogress* threshold is customizable by the user because this value is sent by the Launcher. The value used in Algorithm 1 is 1.0 plus the *overprogress* threshold provided by the user to the Launcher.

$$performance(t) = \frac{cputime_{current}(t)}{cputime_{desired}(t)} \tag{2}$$

3.3. Description of the components

As shown in the figures above, the proposed architecture is composed of several components. Most of them are well-known software packages and frameworks while others are software tools developed by our research group.

All these components require different configuration files and installation steps, which are customized for the different underlying operating systems supported by the VMs. Therefore, this involves a large number of configuration files. Therefore, to ease the deployment and installation process, Ansible roles and playbooks were used for the sake of maintainability and high reusability. These are configuration files that describe the process of installation, configuration and integration with the selected architecture. Furthermore, to keep the component recipes as generic as possible (to further ease maintenance and reuse), such recipes were coded according to the following principles:

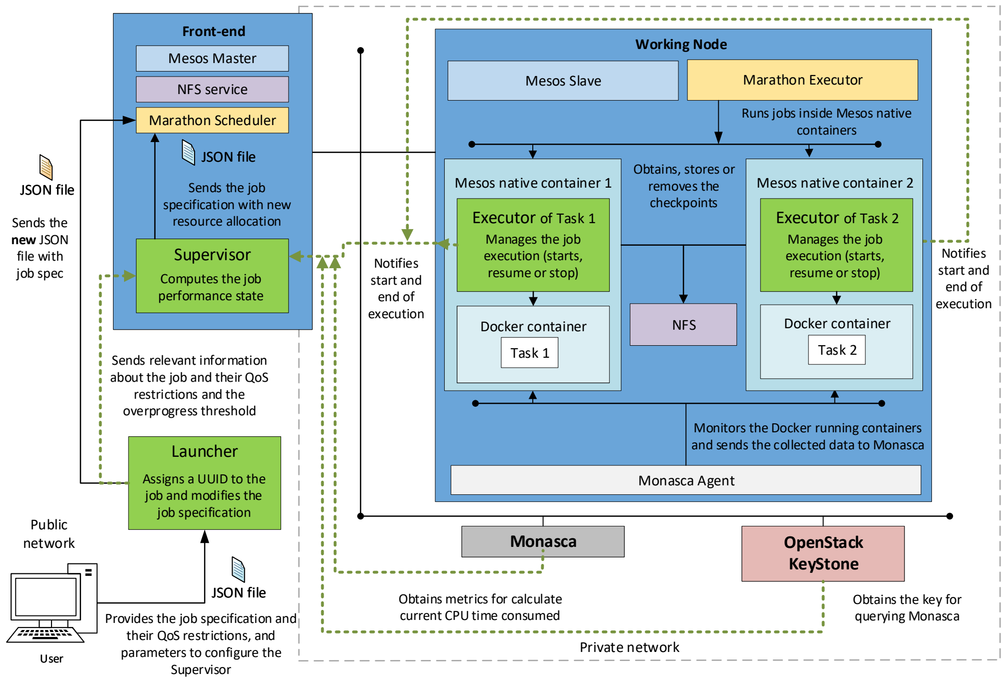
Front-end
Mesos Master
NFS service
Marathon Scheduler
JSON file
Sends the job specification with new resource allocation
Supervisor
Computes the job performance state
JSON file
Sends the new JSON file with job spec
Sends relevant information about the job and their QoS restrictions and the overprogress threshold
Launcher
Assigns a UUID to the job and modifies the job specification
Public network
User
JSON file
Provides the job specification and their QoS restrictions, and parameters to configure the Supervisor
Notifies start and end of execution
Working Node
Mesos Slave
Marathon Executor
Runs jobs inside Mesos native containers
Obtains, stores or removes the checkpoints
Mesos native container 1
Executor of Task 1
Manages the job execution (starts, resume or stop)
Docker container
Task 1
NFS
Mesos native container 2
Executor of Task 2
Manages the job execution (starts, resume or stop)
Docker container
Task 2
Notifies start and end of execution
Monitors the Docker running containers and sends the collected data to Monasca
Monasca Agent
Monasca
OpenStack KeyStone
Obtains metrics for calculate current CPU time consumed
Obtains the key for querying Monasca
Private network

Figure 2: Architecture for vertical scaling.

Algorithm 1: Algorithm used by the Supervisor to obtain the job state.

```
Input : performance, threshold$_{overprogress}$ and threshold$_{underprogress}$
Result: state
begin
    if $performance > threshold_{overprogress}$ :
        state = overprogress ;              /* Decrease resource reservation */
    else if $performance < threshold_{underprogress}$ :
        state = underprogress ;             /* Increase resource reservation */
    else:
        state = ontime ;                                  /* Nothing to do */
end
```

- The production Ansible roles should be stored in public repositories such as GitHub (all the recipes should be open-source and available to the public).
- The variables used inside the Ansible roles should be defined in a way that they can be set up at deployment time. This way, updates can be automatically applied to the roles, so users do not keep outdated configurations on their systems (except those that could have been explicitly modified by the user).
- The Ansible roles should be added to Ansible Galaxy[3] a public repository[4] of roles so that others can reuse them, thus greatly simplifying the roles definition and composition.
- In addition to the Ansible roles, there must exist high-level installation recipes (also stored in GitHub) for the Infrastructure Manager [5] and EC3[6] that contain all the configuration steps for deploying complete virtual infrastructures.
- The recipes should support different platforms (currently Ubuntu 14, Ubuntu 16 and CentOS 7).

Table 1 includes the components that were identified and configured using Ansible roles for the creation of the virtual infrastructure previously described.

The EC3 tool is in charge of deploying the fully configured cluster by using these Ansible recipes. Thus, it provides the required infrastructure with the necessary services to deploy applications by using only a command, that automatically configures and contextualizes all the VMs that compose the cluster infrastructure. As stated above, all the sources of the recipes used to configure the cluster are stored in GitHub[7]. Moreover, for the sake of reproducibility of the results of this contribution, a Docker container image with the EC3 client installed has been released[8].

4. Results

The experiments performed in this paper address three main aspects: i) the efficiency of the deployment of a medium-sized virtual infrastructure; ii) the overhead of the horizontal elasticity compared to the execution with resources deployed upfront; and iii) the ability of the vertical elasticity to reconfigure the reservation of resources for a running job to meet a specific QoS.

For the first case, this work measures the deployment time of a cluster with 50 nodes and 100 processing cores. This time includes the deployment of the VMs, the download and installation of the software dependencies, and the configuration of all the services. This process is entirely automatic.

The second case covers the execution of a set of 20 parallel Spark jobs at different time intervals. Initially, there are no processing resources except the front-end node of

[3]https://galaxy.ansible.com/
[4]https://galaxy.ansible.com/grycap
[5]https://github.com/grycap/im
[6]https://github.com/grycap/ec3
[7]https://github.com/grycap
[8]https://hub.docker.com/r/eubrabigsea/ec3client/

Node Type	**Component**	**Version**	**Requirements**	**Comments**
Front/wn	Apache Mesos	1.4.1	All	Main framework, including Mesos-DNS.
Front	Marathon	1.4.3	1,3,4	For deploying long-term and high-availability services on Mesos.
Front	Chronos	2.1.0	1,2,4	Cron-like job scheduler for Mesos.
Front/wn	Spark	1.6.3	1,5	Execution of Spark code through spark-submit from the Front/End or external resources.
Datanode	Hadoop	2.6	All	HDFS storage backend.
Front/Monasca	Zookeeper	3.4.8-1	All	High availability of Mesos and Marathon.
Front/Monasca/wn	Docker	17.05.0-ce	1-4	Containerization of applications launched through Marathon and Chronos.
Front	Docker registry	2	1-4	Mirroring and caching the Docker images to speed-up distribution along the cluster.
Front	CLUES	2.1.0b	4	Manages horizontal elasticity.
Front	Infrastructure Manager	1.6.6	4	Manages the configuration of the internal nodes.
Front/wn	CRIU	2.6	4	Performs container checkpointing
Monasca/wn	OpenStack Monasca	1.6.1	3,4	Monitoring system, including the Docker plugin. Monasca agent installed in WN.
Keystone	OpenStack Keystone	13.0.0	3,4	A service that provides API client authentication, service discovery, and distributed multi-tenant authorization.
Monasca	Apache Kafka + Storm + Grafana + InfluxDB	2.12, 1.0.2, 4.0.1 & 0.9.5	3,4	Components for the Monasca Server
Front/wn	Weave	2.1.3	All	Provides the overlay network to the container infrastructure.
Front/wn	Vertical Elasticity	1.0	3,4	Proactive vertical elasticity mechanism for Marathon using Monasca

Table 1: Components of the Mesos cluster architecture.

the cluster and the system automatically starts and reconfigures them as required. The execution intervals have been defined in a way that the job queue is flushed completely, triggering the suspending of idle nodes.

The third case deals with the Quality of Service guarantees for a Marathon job executed in the cluster. The cluster is busy with other jobs competing for the resources, so we aim to assign the required CPU time to meet the targeted QoS. The developed tool[9] dynamically readjusts the share of CPU to reduce the assigned resources to a job if it is over progressing and increases the assigned resources if the job progress is lower than expected. We measure the job performance as the amount of CPU time consumed by a job according the given time frame of the QoS agreed.

For the three experiments, the physical infrastructure used is composed by two type of nodes. The first type of node has two Intel(R) Xeon(R) CPU E5-2683 v3 2.00GHz (14 cores) processors, 64 GB of memory RAM, 240 GB of Solid State Disk, two 1 GB Ethernet network adapter and one 10 GB Ethernet network adapter. The second type of node has two Intel(R) Xeon(R) CPU E5-2660 v4 2.00GHz (14 cores) processors, 128 GB of memory RAM, 250 GB of Solid State Disk, two 1 GB Ethernet network adapter and one 10 GB Ethernet network adapter. The Storage Area Network is a Dell Equallogic PS4210 with 16 TB availables. This hardware is managed by the OpenNebula Cloud Management Framework and the KVM hypervisor.

4.1. Deployment metrics

The deployment of the data analytics cluster is done automatically through EC3. The following metrics have been evaluated to show the performance and the impact of the usage of the elasticity on the user experience when interacting with the virtual cluster:

- Deployment of the static components (not managed by CLUES): Mesos master (front-end of the cluster, with 4 CPUs and 16Gb of RAM), Monasca master (with 2 CPUs and 8Gb of memory RAM) and 3 HDFS datanodes (with 2 CPUs and 2Gb of memory RAM). This set of nodes are deployed by the EC3 client in the virtual cluster creation step.

- Deployment of the first working node (with 1 CPU and 2Gb of RAM), including the creation of the *golden image* that will be used to speed up the deployment of the rest of the elastic nodes. This feature consists on creating a VMI from the first working node correctly configured and integrated in the system. Thus, this VMI is used for the next working nodes deployed in the system, accelerating their configuration.

- Deployment of a second working node using the created *golden image* to measure the impact of the usage of *golden images* in subsequent nodes.

- Concurrent deployment of multiple concurrent working nodes (10, 20, 35 and 50 nodes). It will show how the system will react when a large set of nodes are requested.

[9]https://github.com/eubr-bigsea/vertical_elasticity

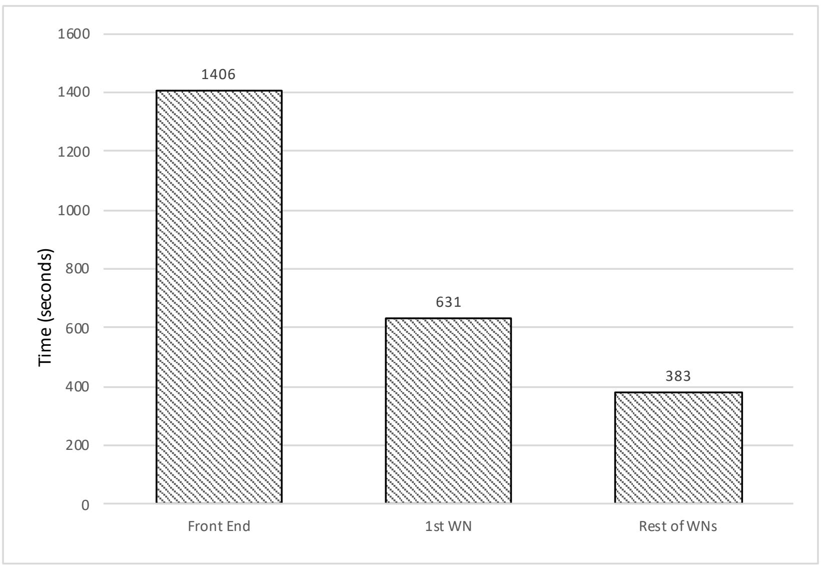

Figure 3: Deployment time of the different node types.

These measures give information about the overheads on the deployment of the full operational cluster and their reconfiguration, which serves as basis for defining elasticity mechanisms and suitable applications.

The first step is the deployment of the front-end and the set of static nodes (front + monasca node + 3 datanodes). The average deployment time for the complete initial infrastructure is 23min 26sec (1406 s). Figure 3 shows the comparative deployment time of the initial infrastructure and the working nodes. The deployment of a node without golden image plus the creation of the golden image when the node has been configured takes an average time of 10min 31sec. (631 s) whereas the deployment of nodes with golden images takes an average time of 6min 38 sec. (383 s). Clearly, nodes deployed from a *Virtual Machine Golden Image*, created on the fly, show a smaller configuration time.

Finally, for testing the scalability of the system, we present the deployment times for a concurrent deployment of multiple nodes (10, 20, 35 and 50 nodes). Table 2 depicts the results of each test, where golden images have been used to accelerate the deployments. The configuration system has been improved in the frame of the EUBra-BIGSEA[10] project to deal with the bottlenecks that appear when a large number of nodes are simultaneously configured. In the original approach a single VM is selected as the "master". Then, Ansible is installed in this VM, which configures all the VMs in parallel. In this new approach (suitable for a large number of simultaneous VM deployments), Ansible is installed in all the VMs and each one configures itself in parallel. This approach increases the latency but reaches higher scalability, being successfully demonstrated in more than 100 machines.

[10]http://www.eubra-bigsea.eu/

Number of nodes	**10**	**20**	**35**	**50**
Average time per node (sec.)	513.647	560.349	666.997	900.841
Total time (sec.)	520.472	592.602	751.997	1052.883

Table 2: Deployment time for different quantities of nodes (deployed simultaneously).

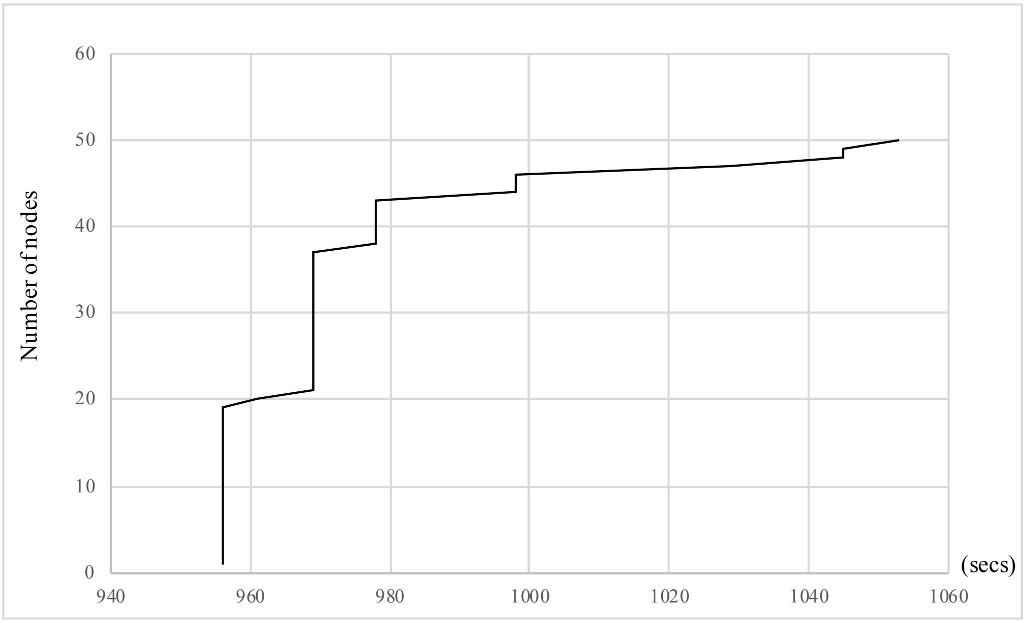

Figure 4: Deployment time for 50 simultaneous nodes.

Figure 4 shows the latency time (in seconds) from the request of the deployment of 50 simultaneous nodes in the cluster to the actual provisioning of the resources. The graph shows the number of machines deployed at each timestep. The figure shows that most of the nodes (42) are fully configured in less than 980 seconds. The rest of the nodes take a bit more time (72 seconds). It is only 7.3% more than the first groups of nodes. This delay is produced by different bottlenecks of the cloud platform (mainly network) when a large number of nodes are configured in parallel.

4.2. Horizontal Elasticity

The second case consisted of submitting 20 parallel data analytics jobs (implemented in Spark) to an infrastructure that initially had only two nodes started (2 vCPUs and 4 GB RAM each). These jobs were submitted at different time frames as shown in Table 3. The infrastructure had to detect the registration of a Spark framework, realize that there are not enough resources and deploy an additional node when a job remains queued longer than a given threshold (5 seconds in the experiment), with a cooling time (waiting time to perform a new action) of 5 minutes. Jobs were prepared to run for approximately

Job	**Submit**	**Job**	**Submit**	**Job**	**Submit**	**Job**	**Submit**
1	0:00:30	6	0:35:54	11	0:55:14	16	1:31:14
2	0:01:15	7	0:52:34	12	1:28:34	17	1:31:54
3	0:01:53	8	0:53:14	13	1:29:14	18	2:05:14
4	0:18:34	9	0:53:54	14	1:29:54	19	2:21:54
5	0:19:14	10	0:54:34	15	1:30:34	20	2:38:34

Table 3: Scheduling of the jobs to be executed.

11 minutes and were able to use up to 4 cores each and request 0.5GB of memory RAM. It is important to state that even if the job requests for 4 cores and Mesos offers it 2 cores, the job would anyway start. If there are enough resources (4 free cores), the job will take them all.

During the execution of the jobs, we measured the timestamp at the submission, execution start and execution end. We also registered all the changes in the status of the nodes, which could be OFF (not deployed), RESTART (being restarted), IDLE (powered-on and without jobs allocated), USED (executing jobs) or SUSPEND (being suspended). Figure 5 shows the results of the evolution of jobs and Figure 6 shows the status of the nodes along time.

Figure 5 shows the number of jobs (vertical axis) along time (horizontal axis) for five metrics. Submitted, Started and Ended lines denote the accumulated number of jobs that have been submitted, have actually started and have been completed over time. The lines Queued and Running denote the number of jobs that are queued or concurrently running at a given time. The execution profile has been defined to ensure that there are peaks of workload that require starting up new VMs and idle periods long enough to trigger the suspension mode for the VMs. This is used to analyze the behaviour system when adjusting the infrastructure. More details are provided in Figure 6.

As depicted in Figure 5, the length of the queue does not grow above one job. The delay between the submission and the start of a job (the difference in the horizontal axis between submitted and started lines) is negligible. It is important to remark that this overhead relates mainly to the time required for the VMs to change from suspended to running, as the VMs are suspended on disk rather than destroyed. It should be pointed out that the execution time for the jobs varies according to the resources available at the executing time.

Figure 6 shows how the working nodes are started and suspended on demand. The figure shows the number of working nodes (vertical axis) that are in each one of the five possible status (described at the beginning of this section) along time (horizontal axis). It is important to outline that transitions are very short, and the submission pattern of the execution enables emptying the queues and triggering the suspension of idle resources. Moreover, the default amount idle time to switch off a node in CLUES was used (20 minutes) but this value can be modified by user depending on the requirements.

4.3. Vertical elasticity

Running batch jobs that need to deal with QoS restrictions on infrastructures with a significant amount of free resources is not a complex task. For this type of infrastructures,

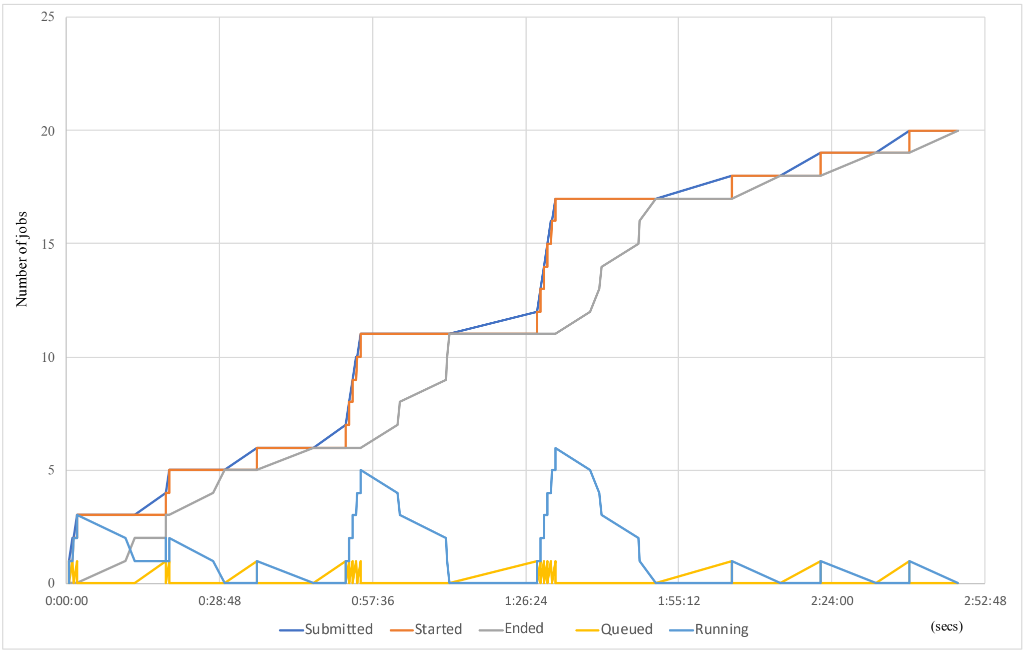

Figure 5: Jobs queued vs jobs running in the platform during the experiment and accumulated list of jobs (suspend mode).

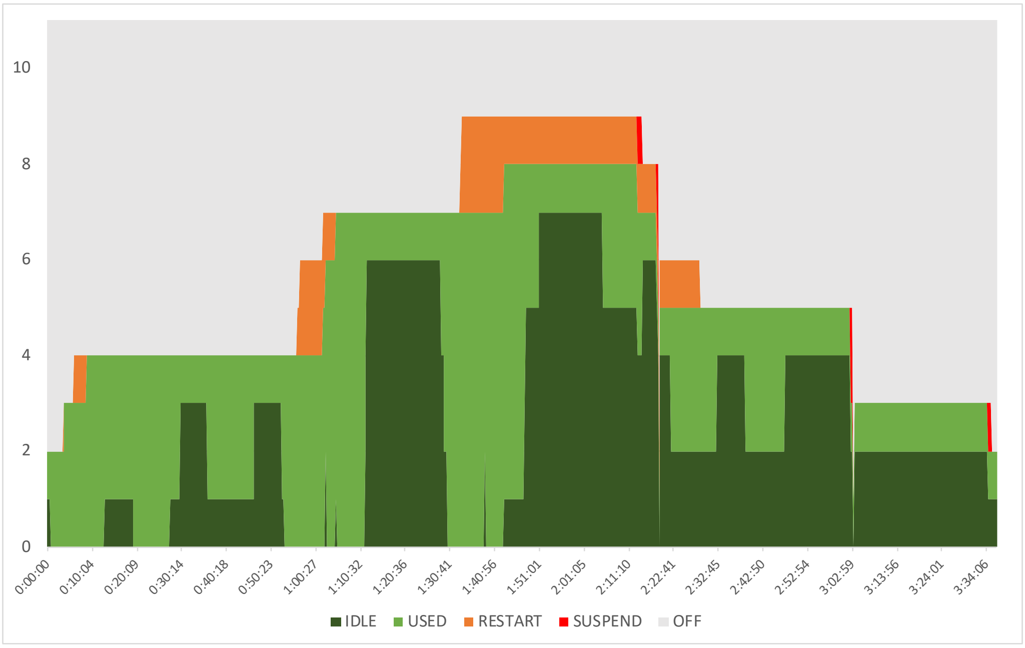

Figure 6: Evolution of the state of the cluster nodes, started and suspended on demand.

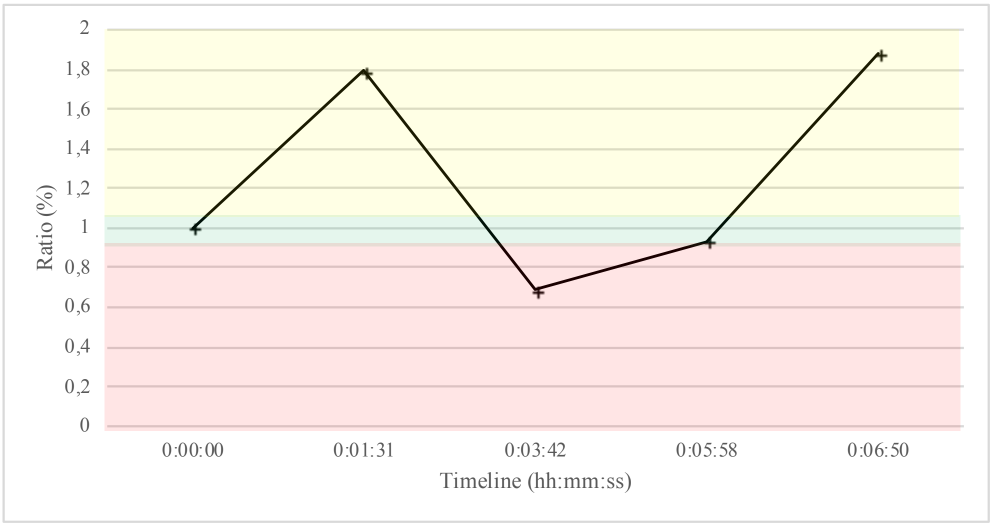

Figure 7: *Performance* during the experiment.

it is very easy to accomplish the QoS restrictions because the scheduler only has to assign the maximum resources to all running applications. Thus, vertical elasticity makes sense in congested infrastructures. The QoS restrictions for this type of jobs are defined as the allocation of a minimum CPU time to the job during a given time interval.

The job used for checking the vertical elasticity capabilities of the implemented architecture takes 480 seconds (one CPU) and 360 seconds (two CPUs) to be completed. The number of the seconds (CPU time) that the mechanism must allocate is set to 400 seconds, which is a value between the completion time when using one and two CPUs. If a node has more amount of CPU share available than the job needs, the job uses all amount of CPU share. The QoS restriction on this test is set to the same completion time than when the job is executed using one CPU. Indeed, the developed mechanism for providing vertical elasticity is designed to make all jobs with QoS restrictions of a congested infrastructure to accomplish their agreed QoS. The maximum *overprogress* threshold is 10%, which means the job enters to *overprogress* state when its performance is 10% better than the required for meeting the targeted QoS.

The test has special interest because the job enters into the three possible states (described in section 3.2) during the execution. In addition, the experiment ends in 393 seconds, demonstrating that the system can scale up and down to guarantee a minimum CPU time. The execution trace can be observed in Figures 7 and 8.

Figure 7 shows the *performance* value (obtained using 2) and the *performance state* (obtained using 1) along the experimentation. The Y-axis and X-axis of the Figure 7 represent, respectively, the *performance* and the time when the sample was collected. The Y-axis colored ranges denote the *performance state*: red, green and yellow represent, respectively, *underprogress*, *ontime* and *overprogress*.

Figure 8 shows a comparison between the consumed and the desired CPU time consumed, and the assigned share of CPU at each sample. The X-axis represent the time

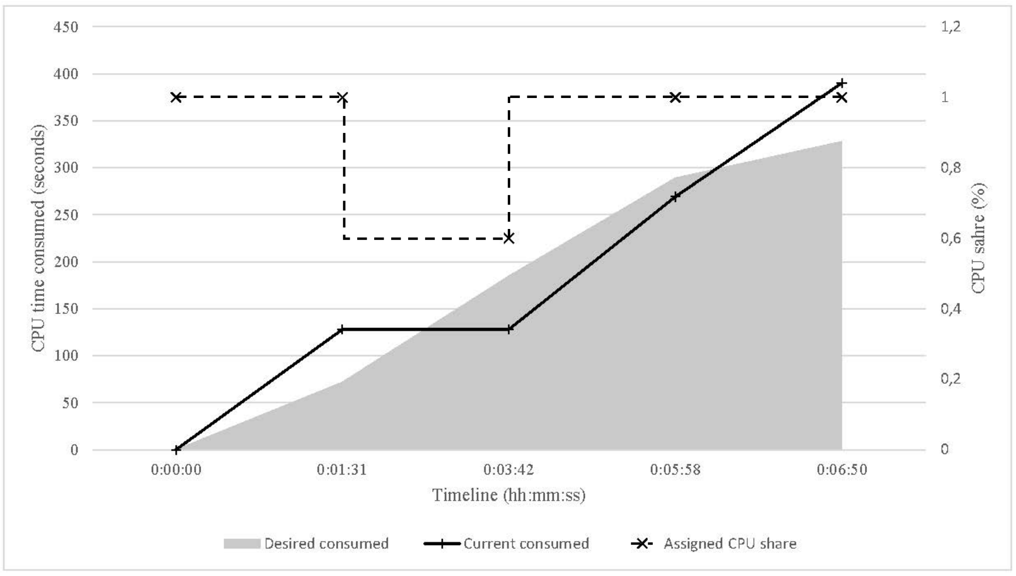

Figure 8: The dashed line represents the CPU share assignation during the experiment. The rest of the figure is a comparison between the CPU time consumed and the CPU time (desired) that the job is expected to consume at the time when the sample was collected.

when the sample was collected. The desired CPU time is the amount of time that the developed mechanism estimates that should be consumed at the moment of which the samples are collected. The left Y-axis represents the amount of CPU time consumed in seconds. The right Y-axis represents the share of CPU assigned to the job during the experiment. If the working node where Marathon executes the job has 1 CPU and the user assigns 1.0 CPU share to the job, it means that the job has the 100% of the CPU share of one CPU. Thus, if the working node has 3 CPU and the CPU share assigned to the job is 2.5 CPUs, the job has reserved all of the CPU share of two CPUs and the 50% of the CPU share of the remaining CPU.

It should be noted that instants when the samples were collected are the same at both figures. The first sample corresponds to the start of the job. When the second sample is collected, it can be observed in Figure 7 that the job *performance state* is *overprogress*. In Figure 8 it can be observed at the second sample that the difference between the current time of CPU consumed and the desired time of CPU consumed is big enough to make the performance above the *overprogress* threshold. Thus, the Supervisor decrements the assigned share of CPU, which can be seen in the dashed line at the same figure. Then, the Executor performs a checkpoint as soon as it realises that the Supervisor decreased the job assigned resources.

As it can be seen in Figure 8, the current CPU time consumed is lower than the desired CPU time consumed in the next two samples but, the job's states are not equal. In case of the third sample, the *performance state* is *underprogress* (the *performance* value is 69%) so the Supervisor will do an increment of the assigned share CPU and the Executor will create a checkpoint.

Regarding the job performance at the fourth sample, the CPU time consumed is lower than the required CPU time calculated using Eq. 1. Due to the *underprogress* threshold is set to 90% and the *performance* value (calculated using the equation 2) is 93%, the *performance state* is *ontime*. For this reason, the job does not require to be resized.

The last sample corresponds with the end of the execution. Figures 7 and 8 show that the application terminates fulfilling the quality of service agreed with 87 seconds of margin.

We measured that the time from the start of the checkpointing to when it is stored in the shared directory (NFS) ranges from 30 to 60 seconds. After several tests, we estimate that executing one checkpointing and restart of the job execution increases the execution time in 6 seconds. Thus, even though the time of checkpointing in NFS is considerable, the downtime of the job execution is negligible. This is because the container continues to be executed while the checkpoint is created, similarly to virtual machine live migration techniques. Indeed, only when the checkpoint is completed the container is stopped and, then, it is immediately rescheduled in a new machine. Therefore, the time required for loading the checkpoint and starting the process is negligible.

It is complicated to estimate the overhead caused by the checkpoint mechanism in the experiment. The duration of the experiment was 393 seconds. The average duration of the job execution with 1 and 2 CPU takes, respectively, 480 and 360 seconds. In this experiment, the mechanism performed two checkpoints, so the overhead of checkpointing is 12 seconds. Thus, the overhead of the mechanism will be lower than 33 seconds and higher than 12 seconds.

5. Discussion

This section compares the proposed tools and solution exposed in this work with the already available solutions that can be found in the literature regarding the execution of time critical applications. Concerning cloud orchestration, our analysis of the state of the art revealed that there is no general orchestration tool that enables the deployment of cloud applications in several on-premises and public IaaS deployments using the standard TOSCA specification. Most of them only provide access to a very limited list of cloud providers. Our proposed cloud orchestration solution, the IM tool, supports the TOSCA standard and a big number of public and federated Cloud providers and on-premises CMPs, making the application cloud agnostic. The IM automates the Virtual Machine Image (VMI) selection, deployment, configuration, software installation, monitoring and update of virtual infrastructures.

The deployment of Big Data frameworks such as Mesos or Kubernetes requires an underlying distributed computing and storage infrastructure, that can be provisioned from on-premises clouds, public clouds or even from bare metal. However, there are several limitations that hinder the adoption of these frameworks, especially by Data Scientists that may be well versed in using the frameworks themselves but not specifically on efficiently deploying and scaling them. The framework presented in this paper in combination with the EC3 tool, considerably simplifies the deployment of these Big Data frameworks. EC3 allows to automatically deploy these Big Data frameworks with a single command, and without the need of user interaction.

Most of the already available solutions to automatically deploy clusters provide a virtual cluster with a fixed number of nodes, other solutions are oriented to a specific

LRMS or they are tied to Amazon EC2 and, therefore, cannot provision nodes from other public Cloud providers, or even on-premises Cloud deployments (e.g. based on OpenNebula, OpenStack, etc.). The vertical elasticity mechanism presented in this work in combination with EC3 and the IM tools allows the user to deploy virtual clusters offering at the same time both horizontal and vertical auto-scaling capabilities. In addition, the contribution of the presented mechanism to the vertical elasticity capabilities (i.e. executing data analytics jobs embedded in Docker containers using Marathon involving common applications with QoS restrictions) was not found in the literature.

6. Conclusions

This paper has presented a software architecture and a set of open-source tools and configuration recipes for deploying a virtual self-managed cluster which offers horizontal (in and out) and vertical (up and down) scalability. Moreover a series of plugins have been developed to offer quality of service capabilities inside the cluster.

Regarding the technical requirements identified at the beginning of the paper, the test cases defined and the results exposed, it can be concluded that all the requirements proposed were fulfilled by the architecture presented. Running unrestricted batch jobs is one of the basic functionalities offered by the standard cluster configuration. In all the test cases it is demonstrated how the cluster admits different types of jobs and executes them without issues.

For running periodic batch jobs, the cluster must be prepared to accept a set of jobs defined to be executed in an specific time. In section 4.2 a batch of jobs are programmed to be launched and the cluster executes them by adjusting the resources available. This demonstrates that the defined architecture is not only able to process this kind of scheduled jobs, it is also able to self adapt horizontally depending on the workload. Moreover, the jobs presented in section 4.2 are a set of Spark jobs that were executed in parallel thus complying with the last requirement presented which required to execute Spark jobs in parallel.

In addition, the QoS restrictions and the vertical elasticity were tested in section 4.3. The execution of batch jobs with QoS restrictions by adjusting the share of CPU assigned is done thanks to a set of plugins developed and deployed automatically in combination with the frameworks available in the architecture presented.

Moreover, and as an extra step towards reusability and community usage, all the code developed for this project is publicly available and any user with access to one of the supported cloud providers (which include the most popular ones) can deploy an elastic cluster and tweak the configuration to fit the needs.

Future work includes testing the cluster with bigger setups, such as several hundred nodes and thousands of jobs during long periods of time but unfortunately and due to all the test being done in real infrastructures with shared resources and real users, the test cases have to be limited.

Acknowledgements

The work presented in this article has been partially funded by the project EUBra-BIGSEA, a project funded by the European Commission under the Cooperation Program, Horizon 2020 grant agreement No 690116. Este projeto é resultante da 3a Chamada

Coordenada BR-UE em Tecnologias da Informação e Comunicação (TIC), anunciada pelo Ministério de Ciência, Tecnologia e Inovação (MCTI) (Europe - Brazil Collaboration of Big Data Scientific Research Through Cloud-Centric Applications). The authors would also like to thank the Spanish "Ministerio de Economía, Industria y Competitividad" for the project "BigCLOE" with reference number TIN2016-79951-R.

References

[1] EUBra-BIGSEA. Deliverable d7.1: End-user requirements elicitation. `http://www.eubra-bigsea.eu/sites/default/files/D7_1_End-User_Requirements_Final.pdf`. [Online; accessed April-2018].

[2] Oasis topology and orchestration specification for cloud applications (tosca). `https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca`. [Online; accessed April-2018].

[3] OpenStack. Openstack heat. `http://wiki.openstack.org/wiki/Heat`. [Online; accessed April-2018].

[4] OpenStack. Heat orchestration template (hot) guide. `https://docs.openstack.org/heat/latest/template_guide/hot_guide.html`. [Online; accessed April-2018].

[5] OpenStack. Openstack. `https://www.openstack.org/`. [Online; accessed April-2018].

[6] OpenNebula. Opennebula. `https://opennebula.org/`. [Online; accessed April-2018].

[7] OpenNebula. Oneflow. `http://docs.opennebula.org/5.2/advanced_components/application_flow_and_auto-scaling/index.html`. [Online; accessed April-2018].

[8] DXC Technology. Eucalyptus cloud platform. `https://github.com/eucalyptus/eucalyptus`. [Online; accessed April-2018].

[9] Amazon. Cloudformation. `https://aws.amazon.com/cloudformation/`. [Online; accessed April-2018].

[10] Cloudify. Cloudify. `https://cloudify.co`. [Online; accessed April-2018].

[11] The Apache Software Foundation. Apache aria tosca orchestration engine. `http://ariatosca.incubator.apache.org`. [Online; accessed April-2018].

[12] I. Giannakopoulos, N. Papailiou, C. Mantas, I. Konstantinou, D. Tsoumakos, and N. Koziris. Celar: Automated application elasticity platform. In *2014 IEEE International Conference on Big Data (Big Data)*, pages 23–25, Oct 2014.

[13] SixSq. Slipstream. `http://sixsq.com/products/slipstream/`. [Online; accessed April-2018].

[14] Sami Yangui, Iain James Marshall, Jean Pierre Laisne, and Samir Tata. CompatibleOne: The Open Source Cloud Broker. *Journal of Grid Computing*, 12(1):93–109, 2014.

[15] University of Stuttgart. Opentosca. `http://www.opentosca.org`. [Online; accessed April-2018].

[16] Amazon. Amazon ec2. `https://aws.amazon.com/ec2/`. [Online; accessed April-2018].

[17] Miguel Caballer, Ignacio Blanquer, Germán Moltó, and Carlos de Alfonso. Dynamic Management of Virtual Infrastructures. *Journal of Grid Computing*, 13(1):53–70, 2015.

[18] Chris Anderson. The end of theory: The data deluge makes the scientific method obsolete. `http://www.wired.com/2008/06/pb-theory/`. [Online; accessed April-2018].

[19] J Dean and S Ghemawat. Simplified data processing on large clusters. *Communications of the ACM*, 51(1):107–113, 2008.

[20] The Apache Software Foundation. Apache hadoop. `http://hadoop.apache.org`. [Online; accessed April-2018].

[21] The Apache Software Foundation. Apache hive. `http://hive.apache.org`. [Online; accessed April-2018].

[22] The Apache Software Foundation. Apache pig. `http://pig.apache.org`. [Online; accessed April-2018].

[23] The Apache Software Foundation. Apache spark. `http://spark.apache.org`. [Online; accessed April-2018].

[24] Ltd. Canonical. Linux containers. `https://linuxcontainers.org/`. [Online; accessed April-2018].

[25] Docker Inc. Docker. `https://www.docker.com/`. [Online; accessed April-2018].

[26] Gregory M. Kurtzer, Vanessa Sochat, and Michael W. Bauer. Singularity: Scientific containers for mobility of compute. *PLOS ONE*, 12(5):1–20, 05 2017.

[27] Wes Felter, Alexandre Ferreira, Ram Rajamony, and Juan Rubio. An updated performance comparison of virtual machines and Linux containers. In *ISPASS 2015 - IEEE International Symposium on Performance Analysis of Systems and Software*, pages 171–172, 2015.

[28] The Linux Foundation. Kubernetes. `http://kubernetes.io/`. [Online; accessed April-2018].

[29] Deis. Deis. https://deis.com/. [Online; accessed April-2018].

[30] The Apache Software Foundation. Apache mesos. http://mesos.apache.org/. [Online; accessed April-2018].

[31] Inc. Mesosphere. Chronos. https://mesos.github.io/chronos/. [Online; accessed April-2018].

[32] Inc. Mesosphere. Marathon. http://mesosphere.github.io/marathon/. [Online; accessed April-2018].

[33] Amazon. Aws auto scaling. https://aws.amazon.com/es/autoscaling/. [Online; accessed March-2018].

[34] Paul Marshall, Kate Keahey, and Tim Freeman. Elastic site: Using clouds to elastically extend site resources. In *CCGrid 2010 - 10th IEEE/ACM International Conference on Cluster, Cloud, and Grid Computing*, pages 43–52, 2010.

[35] Paul Marshall, Henry Tufo, Kate Keahey, David La Bissoniere, and Matthew Woitaszek. Architecting a large-scale elastic environment: Recontextualization and adaptive cloud services for scientific computing. In *ICSOFT 2012 - Proceedings of the 7th International Conference on Software Paradigm Trends*, pages 409–418, 2012.

[36] Massachusetts Institute of Technology. Starcluster. http://web.mit.edu/stardev/cluster/. [Online; accessed April-2018].

[37] Massachusetts Institute of Technology. Starcluster elastic load balancer. http://web.mit.edu/stardev/cluster/docs/0.92rc2/manual/load_balancer.html. [Online; accessed April-2018].

[38] Nikolaos Chalvantzis, Ioannis Konstantinou, and Nektarios Kozyris. BBQ: Elastic MapReduce over cloud platforms. In *Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017*, pages 766–771, 2017.

[39] Amanda Calatrava, Eloy Romero, Germán Moltó, Miguel Caballer, and Jose Miguel Alonso. Self-managed cost-efficient virtual elastic clusters on hybrid Cloud infrastructures. *Future Generation Computer Systems*, 61:13–25, 2016.

[40] Carlos De Alfonso, Miguel Caballer, Fernando Alvarruiz, and Vicente Hernández. An energy management system for cluster infrastructures. In *Computers and Electrical Engineering*, volume 39, pages 2579–2590, 2013.

[41] Z Shen, S Subbiah, X Gu, and J Wilkes. CloudScale: elastic resource scaling for multi-tenant cloud systems. *Proceedings of the 2nd Symposium on Cloud Computing*, pages 5:1—-5:14, 2011.

[42] EUBra-BIGSEA. Deliverable d3.4: Qos infrastructure services intermediate version. https://www.eubra-bigsea.eu/sites/default/files/D3.4%20EUBra-BIGSEA%20QoS%20infrastructure%20services.pdf. [Online; accessed April-2018].

[43] Germán Moltó, Miguel Caballer, and Carlos De Alfonso. Automatic memory-based vertical elasticity and oversubscription on cloud platforms. *Future Generation Computer Systems*, 56:1–10, 2016.

[44] Soodeh Farokhi, Pooyan Jamshidi, Ewnetu Bayuh Lakew, Ivona Brandic, and Erik Elmroth. A hybrid cloud controller for vertical memory elasticity: A control-theoretic approach. *Future Generation Computer Systems*, 65:57–72, 2016.

[45] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Autonomic Vertical Elasticity of Docker Containers with ELASTICDOCKER. In *IEEE International Conference on Cloud Computing, CLOUD*, volume 2017-June, pages 472–479, 2017.

[46] Chao Zheng, Ben Tovar, and Douglas Thain. Deploying high throughput scientific workflows on container schedulers with makeflow and mesos. In *Proceedings - 2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing, CCGRID 2017*, pages 130–139, 2017.

[47] DC/OS. Autoscaling with marathon. https://dcos.io/docs/1.7/usage/tutorials/autoscaling/. [Online; accessed April-2018].

[48] Google. Google cloud. https://cloud.google.com/. [Online; accessed April-2018].

[49] Microsoft. Azure. https://azure.microsoft.com. [Online; accessed April-2018].

[50] Monasca. Monasca. http://monasca.io/. [Online; accessed April-2018].

[51] Apache Software Foundation. Apache kafka. https://kafka.apache.org/. [Online; accessed April-2018].

[52] Apache Software Foundation. Apache storm. http://storm.apache.org/. [Online; accessed April-2018].

[53] Inc InfluxData. Influxdb. https://www.influxdata.com/. [Online; accessed April-2018].

[54] Grafana Labs. Grafana. https://grafana.com/. [Online; accessed April-2018].

[55] OpenStack. Openstack keystone. https://wiki.openstack.org/wiki/Keystone. [Online; accessed April-2018].

[56] Virtuozzo. Criu. https://criu.org/Main_Page. [Online; accessed April-2018].

[57] weaveworks. Weave. `https://www.weave.works/`. [Online; accessed April-2018].