Automatic Elasticity in OpenStack[*]

Leander Beernaert, Miguel Matos, Ricardo Vilaça, Rui Oliveira
High-Assurance Software Laboratory
INESC TEC & Universidade do Minho
Braga, Portugal
lbb@lsd.di.uminho.pt, {miguelmatos,rmvilaca,rco}@di.uminho.pt

ABSTRACT

Cloud computing infrastructures are the most recent approach to the development and conception of computational systems. Cloud infrastructures are complex environments with various subsystems, each one with their own challenges. Cloud systems should be able to provide the following fundamental property: elasticity. Elasticity is the ability to automatically add and remove instances according to the needs of the system. This is a requirement for pay-per-use billing models.

Various open source software solutions allow companies and institutions to build their own Cloud infrastructure. However, in most of these, the elasticity feature is quite immature. Monitoring and timely adapting the active resources of a Cloud computing infrastructure is key to provide the elasticity required by diverse, multi-tenant and pay-per-use business models.

In this paper, we propose Elastack, an automated monitoring and adaptive system, generic enough to be applied to existing IaaS frameworks, and intended to enable the elasticity they currently lack. Our approach offers any Cloud infrastructure the mechanisms to implement automated monitoring and adaptation as well as the flexibility to go beyond these. We evaluate Elastack by integrating it with the OpenStack showing how easy it is to add these important features with a minimum, almost imperceptible, amount of modifications to the default installation.

Categories and Subject Descriptors

C.1.4 [**Parallel Architectures**]: Distributed architectures; C.2.4 [**Distributed Systems**]: Distributed applications; C.4 [**PERFORMANCE OF SYSTEMS**]: Reliability, availability, and serviceability; D.1.3 [**Concurrent Programming**]: Distributed programming

*This work is funded by FEDER through the Programa Operacional Fatores de Competitivdade - COMPETE and by Fundos Nacionais through FCT - Fundação para a Ciência e Tecnologia in the scope of the project Stratus/FCOMP-01-0124-FEDER-015020.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SDMCMM'12, December 3-4, 2012, Montreal, Quebec, Canada.
Copyright 2012 ACM 978-1-4503-1615-6/12/12 ...$15.00.

General Terms

Design, Management, Measurements

Keywords

Cloud Computing, Middleware, Adaptation, Elasticity, Scalability, Automatization

1. INTRODUCTION

Cloud Computing has become an increasingly active topic. The illusion of a virtually infinite computing infrastructure, pay-per-use model and resource sharing among other projects are a few characteristics which make this area so attractive. Fundamental to the Cloud paradigm, and closely tied to the pay-per-use model, is the premise of automatic elasticity [16, 18, 5], in other words, the ability to rapidly increase or decrease resources on demand. Still, most existing Cloud Computing solutions can't provide this feature due to unresolved issues with scalability [10].

There are a few open source software solutions which allow companies and institutions to set up a Cloud infrastructure, also known as Infrastructure as a Service (IaaS). IaaS is the delivery of hardware (server, storage and network) and associated software (operating system virtualization technology and file system) as a service. The provider does very little management other than keeping the data center operational, and the users must deploy and manage the software as they would in their own data center [5]. The deployment is achieved through creating virtual machines which will run on the hardware supplied by the provider. These virtual machines can also be referred to as instances.

Eucalyptus [8], OpenNebula [12] and OpenStack [14] are three popular IaaS projects. Each of these offer the means to manage the life cycle of instances, the underlying infrastructure and user access, among others. Whilst some of these systems do not offer features which allow users or administrators to monitor instance activity, none of them have any form of automated elastic behavior. Our goal is to fill the gap present in these types of software in terms of monitoring and adaptation with a Cloud infrastructure agnostic framework. We decided to concentrate on OpenStack due to the recent growth in its adoption and popularity of the project. It has received contributions from various organizations, such as NASA, RackSpace, Canonical, Dell, Citrix and VMWare.

As the project is relatively recent (about one year and a half since its first version) some areas are still very immature or in specification. Yet, the project shows great potential, as

can be seen by the quantity of new adoptions, ranging from enthusiasts to companies, such as HP [1]. At its current version, Essex 2012.1.2, the elasticity feature is still in an embryonary phase. Despite supporting elastic operations (e.g.: allocation of more resources to an instance), these need to be performed manually by the administrator. There is no way for OpenStack to make decisions or allocate resources without human intervention. This can be attributed to the lack of monitoring and adaptation mechanisms.

In this paper, we describe the implementation of a monitoring and adaptation component, Elastack. It monitors the state of the instances and collects data, which can later be exported. The collected values are then used to manage the infrastructure. By using standard technologies, Elastack can easily be integrated with various Cloud infrastructures, providing elasticity as well as other management features (e.g.: monitoring and billing).

The remainder of this paper is organized as follows: in Section 2 we introduce OpenStack, followed by a description of Elastack's architecture in Section 3. Section 4 covers the implementation process of Elastack and Section 5 presents our experimental results. Finally, Section 6 presents related work, and Section 7 concludes this paper.

2. OPENSTACK

OpenStack can be regarded as the open source version of Amazon's EC2 and S3 services since it has an API that is compatible with those used by Amazon's services, besides providing its own API [9]. The main reason for this is to ease the porting of existing projects to its infrastructure.

OpenStack is composed of various components: Compute (Nova), Object Storage (Swift), Identity (Keystone) and Image Service (Glance). To create a minimum Cloud infrastructure, we require only Nova, Keystone and Glance services which we describe below. The interaction between all of the components can be seen in Figure 1. Swift is an elastic storage service, similar to Amazon's S3 [3] and can be used optionally.

Nova

The Compute component, Nova, is in charge of all the procedures required to sustain an instance's lifecycle: computational resource management, networking, authorization and scalability. Nova, on its own, does not have any virtualization features. It resorts to the livbirt APIs to fill this gap.

Queue Server Is the AMQP protocol communication service used by all of OpenStack's components. All communications happen asynchronously to avoid that long operations block while waiting for the result (e.g.: uploading a new image).

Nova API Is the endpoint through which the clients and administrators can operate on OpenStack. It is in charge of translating the received operations to tasks which will be executed by each of OpenStack's components.

Nova Compute Manages the lifecycle of each instance. In a production environment, it is usual to encounter several of these. The compute node where an instance is launched depends on the policy in use by the scheduler.

Nova Network Is in charge of managing network configurations in the infrastructure, as well as assigning IP addresses to the instances and the automatic configurations of VLANs.

Nova Scheduler This component decides where an instance should be launched. The decision is based on the current policy in use by the scheduler and can be based on factors such as memory, system load, physical distance, CPU architecture, among others.

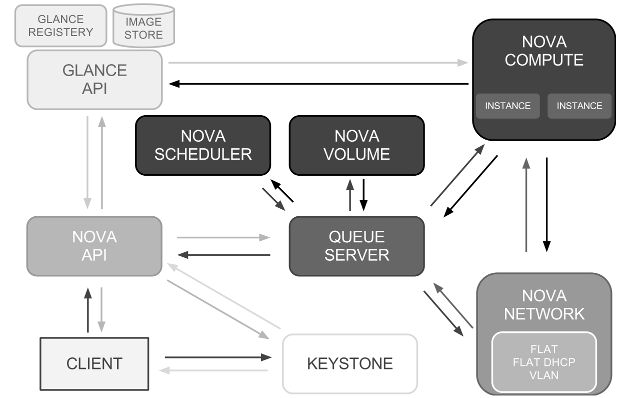

Figure 1: Illustration of the OpenStack architecture.

Nova Volume Offers persistent storage for an instance. By default, when an instance is terminated, the allocated disk space disappears and all the information contained there is lost. Through this component, it is possible to assign persistent storage to an instance.

Keystone

Keystone is the central identity and authentication service used by all the services provided by OpenStack. This component also acts as a discovery point for all the services present in the infrastructure. This information is obtained after a successful authentication.

Glance

Glance is responsible for the storage and availability of the operating system images to be used by the instances. These images can be stored on the local file system, on the host where Glance is running, or stored in Swift for higher availability.

3. ELASTACK

In order to bridge the gap in terms of adaptability and monitorization in OpenStack, we propose the architecture depicted in Figure 2. One of the objectives of our architecture is to be scalable with the underlying infrastructure. As such, we install a monitoring daemon on each Nova Compute node, which will monitor the instances running on that node. The collected information is then made available for consumption through a JMX service.

Serpentine [11] is an adaptive middleware which allows a service/system to adapt to changes that might occur in a production environment without human intervention. Since it was designed to be scalable, its components do not depend on a persistent state and therefore can be organized in a hierarchy, thus allowing a micro and macro-management of the infrastructure. Communication occurs through JMX and the control logic is defined through scripts provided by the administrator. To do so, it resorts to Java's scripting engine in order to allow the execution of various scripting languages (JavaScript, Python, Ruby, among others). These scripts define the management policies to be applied to the

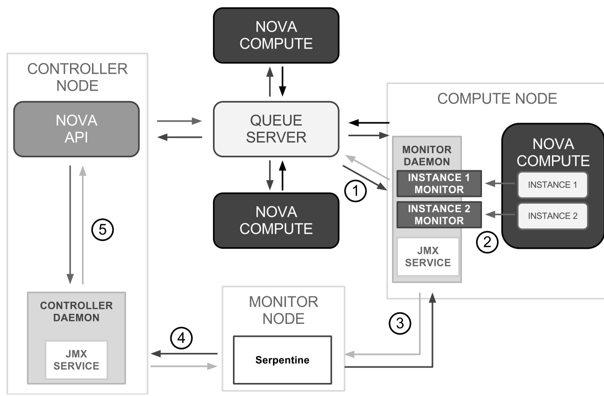

Figure 2: Illustration of the Elastack architecture.

system. The use of script can also help mask complex tasks by abstracting them with functions supplied during the execution of the script.

Our architecture is made out of three fundamental components: *monitor daemons*, *controller daemon* and the *serpentine script*. The monitor daemon is a background process which will run on each nova-compute node in order to scale with the rest of the infrastructure. Internally, it will monitor each instance running on that node and periodically update the collected data for all monitored instances.The monitoring is performed by an *instance monitor*, as seen in (2) in Figure 2. To determine when an instance is launched or terminated, it listens to the events propagated by the Queue Server (1).

Each of these monitors export a MBean which can be accessed through the JMX service. With these MBeans, we can directly access the latest state collected from each instance. However, we can only do this if we know on which host the instance resides. To circumvent this problem, the monitor daemon itself exports a MBean through which we can retrieve information about the physical host, query the daemon to determine which instances are being monitored, and access the instance's information.

The controller daemon is another background process which, ideally, should be run on the same host as the controller node. However, it is also possible to run this service elsewhere. Its main function is to abstract the underlying Cloud service and to isolate the remaining Elastack components from it, thus ensuring that it remains Cloud infrastructure agnostic. In our case, the controller daemon exports a set of methods through a MBean which allow us to create and terminate instances in OpenStack. This is achieved by communicating with the nova-api process on the controller node (5).

Finally, the Serpentine script defines the behavior the system should have. First, a configuration file needs to be supplied in order to connect to all the existing monitor daemons and the controller daemon (3,4). These will then be available during the script's runtime. With the previously described data, the administrator can now define the behavior of the system, which can range from elastic configurations to a simple activity monitor.

Since Elastack operates on an instance basis, horizontal scalability, and not on an application basis, it should be noted that in order to achieve a true elastic behavior, the applications installed on the instances being managed should be ready to enter and exit at any point without leaving the application in an inconsistent state. With some extra work, it is also possible to have the script coordinate with the application when it should initiate or terminate an instance, as we shall see in Section 5.

4. IMPLEMENTATION

In order to monitor each instance, OpenStack provides a command entitled "diagnostics". Sadly, this method is only available for the XEN hypervisor through the XEN API. However, we are using a setup with libvirt, since this hypervisor allows OpenStack to manage a greater number of hypervisors with the same API, including XEN. So, we had to extend OpenStack to support the method with the libvirt virtualization library. The resulting code has been submitted and accepted into OpenStack and will be available starting with the Folsom-3 milestone[1]. Unfortunately, since there is no specification regarding the results of the command, we have no means to assure that the current output produced by this command will be valid in the future releases. Therefore, we retrieve our data directly from the libvirt daemon running on the compute nodes. This in turn enables us to collect more data than the one supplied by the "diagnostics" command.

To monitor the instances we need to know when they are created and terminated. OpenStack offers a notification system which has a series of events that describe the activity of an instance [15][2]. The monitor daemon then subscribes to these events, and registers the creation and termination of each instance. Each time an instance is created, it is put in a list of instances to be monitored. Periodically, the daemon traverses the list and collects the state of each of these instances. The last collected state is maintained in memory until the instance terminates. The full collection of collected states are written to a Swift container for future access. The stored information could be useful for billing, behavior analysis and traceability.

As mentioned previously, the collected data is made available through a MBean exported by the monitor daemon. This MBean provides methods to access information regarding the host machine, the number of instances running and CPU, memory, network traffic and disk usage of each instance. For simplicity's sake, we only export basic data types through JMX. Thus, most methods require an identifier of the instance or a key representing a component (e.g.: interface name, disk name) to be supplied if we are accessing instance data through the monitor MBean.

At the moment, the Controller MBean contains a set of methods which allow instances to be created or terminated. Each of these requests invoke a Python script which uses the official OpenStack API to carry out each operation.

Although the implementation above is targeted specifically at OpenStack's infrastructure, other platforms can be targeted with little work. There are two possible approaches, implement the interfaces present in the monitor and the con-

[1]For more details please see commit ad54ed53cf6a475ad0f8042f8b95454a8c0b35a4 on `https://github.com/openstack/nova/`

[2]These notifications are disabled by default. To enable them it is necessary to add `notification_driver = nova.notifier.rabbit_notifier` to the nova configuration file.

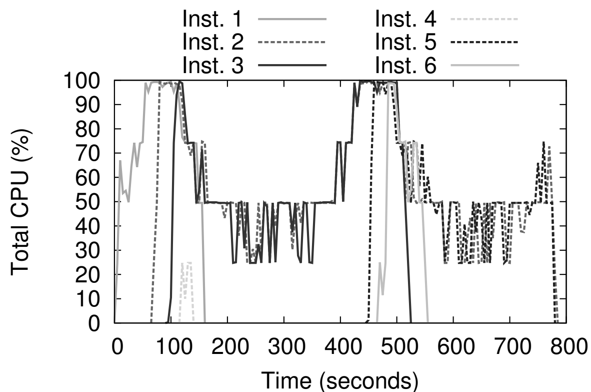

Figure 3: Evolution of the instance's CPU load.

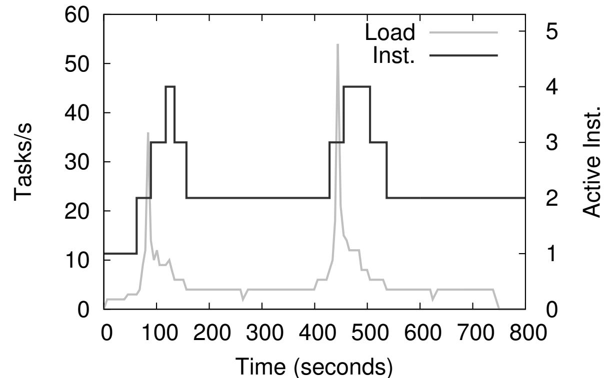

Figure 4: Evolution of the number of instances active and the number of tasks launched per second.

troller or implement a new MBean with the same method set. The first approach reuses most of the existing logic in the monitor and controller daemon, only platform specific methods need to be added in order to retrieve the list of running instance and information collection. The latter allows the monitor and controller deamon to be swapped out with other implementations. These implementations are only required to export the same MBean methods to maintain compatibility with the Serpentine script.

5. EXPERIMENTAL EVALUATION

To test our proposed architecture, we prepared an installation of OpenStack with four nova-compute nodes and one controller node. Each of the machines is equipped with a i3-2100 processor clocked at 3.10GHz, 4GB RAM and a 250 GB SATA II hard drive. We intend to demonstrate how the system would react to a sudden increase in workload. When the workload rises, new instances need to be created to accommodate the new requests, and when the workload decreases, instances are removed in order to avoid wasting unnecessary resources. Each instance will be launched with 4 CPU cores and 3GB of RAM.

To simulate the workload, we developed a small load balancer written in Java which distributes a collection of tasks throughout the running instances. Each instance is preconfigured to run the client process which will receive the tasks from the load balancer. Each task executes the `stress` command for one CPU core during 22 seconds. In order to detect new instances and to avoid terminating instances which still have tasks running, the Serpentine script coordinates the creation and termination of instances with the load balancer. For the sake of simulating a variable workload, the load balancer sleeps for a certain amount of seconds before distributing a new set of tasks to the instances. A greater sleep time will result in a reduced workload.

When an instance is created by Serpentine, it will inform the load balancer of such. The load balancer will then attempt to connect to the client running on that instance. When a connection is made, it will start distributing tasks to this instance. Before terminating an instance, Serpentine first informs the load balancer that the instance is marked for termination. The load balancer will then cease to distribute tasks to that instance and, when all pending tasks have finished executing, mark it as ready to be terminated. In the meantime Serpentine will query the load manager for the termination status of the instance. Once a positive reply is given, Serpentine terminates the instance. New instances are created when $x \geq (90.0 \times n)$, where x is the combined CPU usage of all instances and n the number of instances. Instances are removed when the total CPU usage can be distributed throughout the remaining instances.

In order to make the decision process more adaptable to sudden changes in workload and to avoid sequential decision patterns (such as: create instance, remove instance, create instance) the script supplied to Serpentine uses a simple hysteresis mechanism. It essentially requires that an addition or removal decision is made twice before acting accordingly. To do so, it stores the previous decision at the end of the first execution of the script. When the script runs a second time, the current decision is compared to the previous one. If the decisions are contradictory or no decision is made, the previous decision is rendered invalid.

Figures 3 and 4 represent the average CPU load of each of the instances and the number of tasks launched per second versus the number of active instances, respectively. Each of the measurements are taken in a 15 second interval. The interval at which the measurements are taken is configurable; we assessed that 15 seconds would be the optimal value for this test case. It should be noted that the interval time is inversely proportional to the number of update cycles executed by the monitor. In both figures we can observe that when the CPU load reaches its peak or the number of tasks per second increases, new instances are created. We can also make out a certain delay from the point where the load reaches its peak and a new instance is launched. This is mainly due to two factors: decision time and instance start up time. The first is a consequence of the hysteresis mechanism that requires two complete consecutive executions of the script to reach the same decision. This alone requires an interval of at least 30 seconds. The latter can be attributed to the time it takes for an instance to start and is limited by the hardware in use. This is also visible when the load reduces and instances are removed.

6. RELATED WORK

In terms of monitoring, there are some projects which aim to fill the gap for this feature or provide a way to integrate the monitoring component in a custom management software, such as Zenoss [19]. For instance, the project re-

ferred in [7] registers the total usage of each resource by the infrastructure. This makes it suitable only for billing purposes, since the collected data does not contain suitable information for the adaptation process. Finally, the OpenStack community is working on an official implementation which aims to bring monitoring and billing components to the infrastructure [13]. This solution will build on our contribution and, when finished, will present a standard method to obtain instance and host data as well as retrieve stored data for further reference.

The architecture proposed by [10] is similar to ours. The controller observes the system's load and according to the model in use, equivalent to the scripts in Serpentine, decides how many new instances need to be launched. The controller is also capable of using previously collected data to improve the decisions being made. The system works directly with the XEN API while ours aims to be Cloud agnostic.

The framework CloudScale [17] performs a series of previsions about the behavior of the instance and, based on the collected data, adds resources to or removes resources from the instance. CloudScale also works on the XEN API and acts directly on the instance instead of launching new ones, in other words, it scales that instance vertically. Whilst Elastack is agnostic to underlying hypervisor, it currently only scales the system horizontally. However, a specific script could easily be written to scale an instance vertically, since all the features required for this action are already in place.

Amazon's Auto-Scale [2] is very similar to Elastack's architecture. First, a user must group together a set of similar instances. The Auto Scale will then act on this group. In order for it to work, the user must define a launch configuration for new instances and a set of triggers. Each trigger is composed of a CloudWatch metric, which is triggered when a certain condition is met, and a policy which defines how to handle the metric and when the group should be scaled up or down. Elastack, on the other hand, integrates these parameters all in one script together and is not tied to Amazon's infrastructure.

Finally, Kaleidoscope [6] introduces the concept of micro-elasticity. Instead of launching new instances from scratch to satisfy the system's demand, each instance is cloned with a copy of the complete or partial state of the original instance. Contrary to a normal start in which the operating system needs to start and no information is cached, Kaleidoscope tries to copy most of the information marked as important into the clone. This allows for a new instance to boot up and rapidly respond to requests since it can take advantage of the data provided by the original instance. To determine which sections should be copied, Kaleidoscope installs itself into the machine and communicates with another process residing on the host. The problem with this approach is that it needs to be deeply integrated with the Cloud infrastructure. For instance, in our case, to effectively use this, OpenStack should have knowledge of this framework's existence in order to install it on every launched instance and provide a separate option to perform the cloning operation.

7. CONCLUSIONS

Automatic elasticity, the ability to quickly increase or decrease the resources as necessary, is one of the most desirable characteristics of Cloud infrastructures. Without elasticity, the service providers can neither offer a true pay-per-use payment model nor maximize their resource monetization. While elasticity does not directly increase the providers profit, it certainly offers a more competitive advantage.

In this paper we proposed an adaptation and monitoring component for Cloud infrastructures (IaaS), Elastack. The evaluation of Elastack demonstrated a good adaptability to the submitted load by answering with the increase or decrease in the number of virtual machines in order to satisfy all requests.

Taking into account the design of Elastack's architecture, it can be used with various Cloud infrastructures. Additionally, due to the nature of the exported data and the control flexibility offered by Serpentine, Elastack can be used for a wide variety of tasks beyond providing elasticity and monitoring capabilities. These will in turn better equip the administrator to manage the infrastructure and provide better QoS, resource usage, reduce power consumption and satisfy its costumers' needs.

In terms of future work, Elastack should be adapted in order to be integrated with OpenStack's official monitoring solution [13] as soon as it is available. Statistical machine learning [4] could be added to Elastack to enable dynamic scaling. Finally, a custom DSL could be added in order to aid system administrators in the creation of the control scripts.

8. REFERENCES

[1] http://www.openstack.org/blog/2012/04/openstack-foundation-update/.

[2] Amazon. Amazon auto scale home webpage. http://aws.amazon.com/autoscaling/.

[3] Amazon. Simple storage service. http://aws.amazon.com/s3/. Scalable and reliable storage service.

[4] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A view of cloud computing. *Commun. ACM*, 53(4):50–58, Apr. 2010.

[5] S. Bhardwaj, L. Jain, and S. Jain. Cloud computing: A study of infrastructure as a service (iaas). *International Journal of engineering and information Technology*, 2(1):60–63, 2010.

[6] R. Bryant, A. Tumanov, O. Irzak, A. Scannell, K. Joshi, M. Hiltunen, A. Lagar-Cavilla, and E. de Lara. Kaleidoscope: cloud micro-elasticity via vm state coloring. In *Proceedings of the sixth conference on Computer systems*, EuroSys '11, pages 273–286, New York, NY, USA, 2011. ACM.

[7] G. Dynamics. Nova-billing. https://github.com/griddynamics/nova-billing.

[8] Eucalyptus home page. http://www.eucalyptus.com/. Eucalyptus Cloud Infrastructure.

[9] O. Foundation. Openstack compute api. http://docs.openstack.org/api/openstack-compute/1.1/content/. Complete reference of the OpenStack v1.1 API.

[10] S. J. Malkowski, M. Hedwig, J. Li, C. Pu, and D. Neumann. Automated control for elastic n-tier workloads based on empirical modeling. In *Proceedings of the 8th ACM international conference on Autonomic computing*, ICAC '11, pages 131–140, New York, NY, USA, 2011. ACM.

[11] M. Matos, A. Correia, Jr., J. Pereira, and R. Oliveira. Serpentine: adaptive middleware for complex heterogeneous distributed systems. In *Proceedings of the 2008 ACM symposium on Applied computing*, SAC '08, pages 2219–2223, New York, NY, USA, 2008. ACM.

[12] Opennebula home page. http://www.opennebula.org/. OpenNebula Cloud Infrastructure.

[13] OpenStack. Efficient metering in openstack blueprint. http://wiki.openstack.org/EfficientMetering. Especification of the monitoring project.

[14] Openstack home page. http://www.openstack.org/. OpenStack Cloud Infrastructure.

[15] OpenStack. Openstack system usage data. http://wiki.openstack.org/SystemUsageData. Especification of the eventos triggered by the notification system.

[16] D. Owens. Securing elasticity in the cloud. *Commun. ACM*, 53(6):46–51, June 2010.

[17] Z. Shen, S. Subbiah, X. Gu, and J. Wilkes. Cloudscale: elastic resource scaling for multi-tenant cloud systems. In *Proceedings of the 2nd ACM Symposium on Cloud Computing*, SOCC '11, pages 5:1–5:14, New York, NY, USA, 2011. ACM.

[18] L. M. Vaquero, L. Rodero-Merino, and R. Buyya. Dynamically scaling applications in the cloud. *SIGCOMM Comput. Commun. Rev.*, 41(1):45–52.

[19] Zenoss. Openstack plugin for zenoss. https://github.com/zenoss/ZenPacks.zenoss.OpenStack.