Future Generation Computer Systems 26 (2010) 1226–1240

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

From infrastructure delivery to service management in clouds

Luis Rodero-Merino [a,*], Luis M. Vaquero [a], Victor Gil [b], Fermín Galán [a], Javier Fontán [c], Rubén S. Montero [c], Ignacio M. Llorente [c]

[a] Telefónica Investigación y Desarrollo, Madrid, Spain

[b] SUN Microsystems, Regensburg, Germany

[c] Departamento de Arquitectura de Computadores y Automática, Facultad de Informática, Universidad Complutense, Madrid, Spain

ARTICLE INFO

Article history:
Received 15 May 2009
Received in revised form
4 January 2010
Accepted 25 February 2010
Available online 6 March 2010

Keywords:
Cloud computing
Advanced service management
Automatic scalability

ABSTRACT

Clouds have changed the way we think about IT infrastructure management. Providers of software-based services are now able to outsource the operation of the hardware platforms required by those services. However, as the utilization of cloud platforms grows, users are realizing that the implicit promise of clouds (leveraging them from the tasks related with infrastructure management) is not fulfilled. A reason for this is that current clouds offer interfaces too close to that infrastructure, while users demand functionalities that automate the management of their services as a whole unit. To overcome this limitation, we propose a new abstraction layer closer to the lifecycle of services that allows for their automatic deployment and escalation depending on the service status (not only on the infrastructure). This abstraction layer can sit on top of different cloud providers, hence mitigating the potential lock-in problem and allowing the transparent federation of clouds for the execution of services. Here, we present Claudia, a service management system that implements such an abstraction layer, and the results of the deployment of a grid service (based on the Sun Grid Engine software) on such system.

1. Introduction

Cloud systems [1,2] have recently emerged as a "*new paradigm for the provision of computing infrastructure*" for a wealth of applications [3–5]. Thus, providers of software-based services, which we will denote as *Service Providers* (SP), are freed from the burden of setting up and managing the hardware and/or software platforms required by their services. These resources are provisioned by the cloud platform, offered by a *Cloud Provider* (CP).

Cloud systems are classified by the kind of resources they offer: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). Arguably, IaaS systems are the ones that have raised the greatest interest so far. Many efforts are devoted to find new business models based on these services; they all have a key common feature: they try to offer infrastructure as an utility for SPs to run their software-based systems.

Thanks to IaaS cloud technologies, SPs can quickly arrange new computing infrastructure in a pay-per-use manner. SPs can adaptively allocate virtual hardware resources according to their services' load: if the load grows, new resources (like Virtual Machines, VMs) are demanded to avoid a possible service outage, which would impact on the offered Quality of Service (QoS); when the load shrinks the SP can release idle resources to avoid paying for underused equipment.

However, in spite of the obvious advantages that clouds bring, present IaaS still present important drawbacks. Although the SP's main concern is the *service lifecycle* (deployment, escalation, undeployment), IaaS interfaces are usually too close to the infrastructure, forcing the SP to manage manually the infrastructure (VMs) assigned to the service. This can limit the interest of SPs in cloud solutions. They are willing to reduce costs, but without an excessive administrative burden. Also, many cloud users are concerned about vendor lock-in problems due to the lack of standards that hinder the migration of processes and data among clouds. Hence, there is still room for evolved cloud systems that implement needed, but still unaccomplished, enhancements.

In this paper, we propose a new abstraction layer for cloud systems that offers a more friendly interface to SPs by enabling the control of the services lifecycle. We also introduce *Claudia*, our implementation proposal of such a layer.

The rest of this paper is organized as follows. Section 2 presents existing solutions and useful standards for implementing a certain level of automatic scalability control and their limitations. This section also enumerates the goals to be addressed to enable the

* Corresponding address: Telefónica I+D, C/Emilio Vargas 6, C.P.:28043, Madrid, Spain. Tel.: +34 913374247; fax: +34 913374272.

E-mail addresses: rodero@tid.es (L. Rodero-Merino), lmvg@tid.es (L.M. Vaquero), victor.gil@sun.com (V. Gil), fermin@tid.es (F. Galán), jfontand@fdi.ucm.es (J. Fontán), rubensm@dacya.ucm.es (R.S. Montero), llorente@dacya.ucm.es (I.M. Llorente).

0167-739X/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2010.02.013

automatic scaling of services. Section 3 describes the features and capabilities of Claudia and how it fulfills the challenges above. Section 4 shows the results of the deployment and execution of several different scalability use cases controlled by Claudia. Section 5 indicates ongoing work on some of the challenges identified with Claudia. Finally, Section 6 emphasized the final conclusion of the present work.

2. Background and challenges ahead

Existing IaaS providers propose different alternatives to access to their services, typically based on WSDL or REST protocols: Amazon API [6], GoGrid's API [7],[1] Sun's Cloud API [8] or VMware's vCloud [9] are some examples. Amazon is extending its core APIs to provide higher level services, such as Amazon's Cloud Watch and AutoScale, that aim to automate the scaling process of Amazon-deployed VMs. Also, RightScale manages the creation and removal of VMs according to queues or user-defined hardware and process load metrics [10]. However, its auto-scaling features are still very inflexible, since scalability can only be defined in terms of the variables monitored in the server templates RightScale provides. Hence, the SP cannot use arbitrary service metrics. Also, it is not possible to set bounds depending on business criteria. These same limitations are present in similar systems offering automatic scalability over cloud platforms, such as Scalr [11], WeoCeo [12], etc. In addition, these lower level APIs are defined by individual organizations, which could lead to vendor lock-in problems. Fortunately, initiatives such as the Open Cloud Computing Interface (OCCI) [13] is an active open standard (based on RESTful) with wide industrial acceptance (GoGrid, ElasticHost, FlexiScale, Sun and others). Indeed, some of the paper authors are OCCI promoters and authors of the first implementation of the standard.[2] Yet, OCCI still lacks service-level primitives to be invoked that hide the VM-related operation to a SP.

All the solutions above still lack the ability to handle the lifecycle of services. For instance, in order to automatically grow/shrink the resources used as load varies, they only implement scaling rules based on infrastructure metrics or, at best, load balancer-derived data. Consequently, these APIs are well below the abstraction level required by SPs. They are too close to the machine level, lacking primitives for defining service-relevant metrics, the scalability mechanisms automatically governing the system, definition of applicable Service Level Agreements (SLAs), etc. In conclusion, they do not provide ways for SPs to describe services in a holistic manner, as they do not offer the appropriate abstraction level.

Due to this limitation, it is not possible to deploy services at once in one single step. SPs have to install, customize and manage VMs one by one. Also, as commented above, no CP allows us to define automatic scalability actions based on custom service metrics. Thus, to make full use of the scaling capabilities of clouds, SPs have to monitor the service status constantly in order to allocate new resources (such as VMs) or release unused ones as required. Besides, today no cloud platform supports as yet the configuration of certain business rules, as for example limits on the maximum expenses the SP is willing to pay so she does not go bankrupt due for example to Economic Denial of Sustainability (EDoS) attacks, increasing the resource consumption and associated bill of a given SP.

Thus, to cope with the full automation requirements that SPs demand, cloud platforms must evolve from just infrastructure delivery to automated service management. We identify the four goals that should drive such evolution:

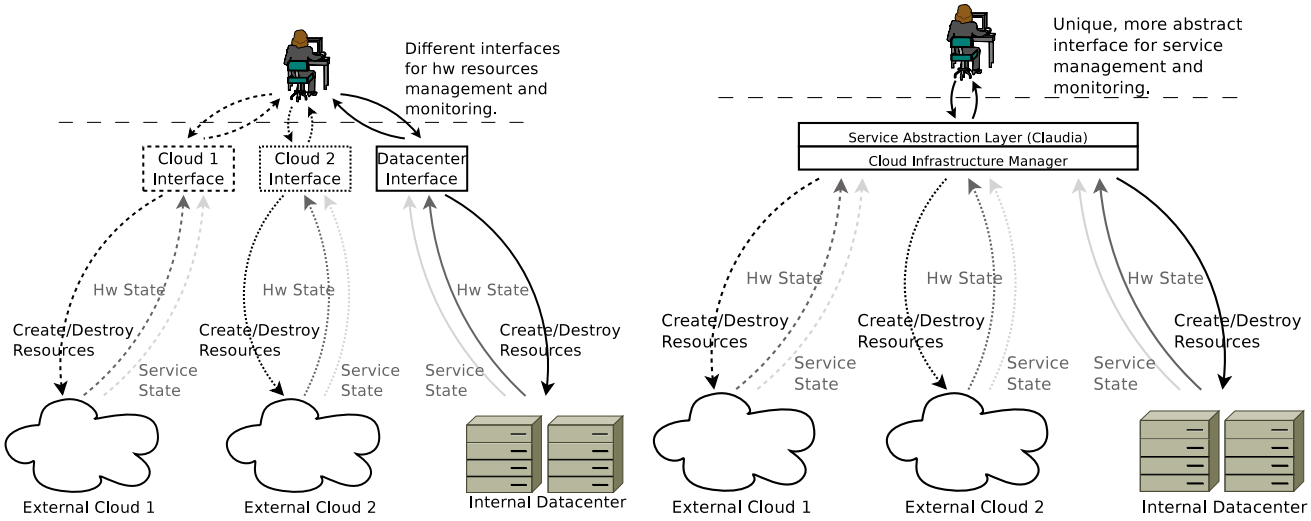
1. *Appropriate Service Abstraction Level* that provides SPs with the friendliness needed to define and manage services, in contrast with the present situation where they have to deal with individual resources (VMs). A service definition shall include relationships among components (e.g. deployment order), and business and scalability rules to be observed. It should be rich enough so even complex services can be deployed automatically by the cloud platform. This implies that the SP has to be allowed to specify, for example, that if a Web Server and a Database are linked together, then the Database must be started first.

   Besides, users shall be able to define how each VM must be customized, so the cloud platform knows which configuration information is to be provided to each VM at start time, and, thus, free the SP from that task. Following the previous example, the Web Server can need the Database location (its IP address) to contact it, but this can be unknown before deployment time.
2. *Automatic Scalability* is required to reduce the management tasks. Full user-defined scalability automation that takes into account *the service status* is a missing feature in current IaaS offerings. The SP will define how the service has to be scaled using her own experience and knowledge of the service and the different factors that can impact on its performance. For example, the SP shall be able to specify that the ratio of database transactions per database replica should neither be greater than 1000 (to avoid overloading), nor less than 100 (to avoid resource overprovisioning that would impact on the total cost).
3. *Smart Scaling* is later required to specify the rules that constrain the automatic scaling. For instance, this feature can prevent malicious users to make a deployed service grow far beyond an acceptable risk/cost, by implementing user-defined or business-based bounds. For example, the SP shall be able to set bounds on the amount of resources provisioned to avoid that automatic scaling actions lead to too high costs. On the other hand, it should be possible for the CP to control the amount of resources that each SP can demand depending on business criteria (e.g. users that delay payments will not be allowed to allocate too many resources).
4. *Avoiding Cloud Vendor Lock-in*: the increasing number of CPs and their heterogeneous interfaces together with the lack of interoperability can lead SPs to be too tied to a particular CP. Instead, services should be able to run on different CPs. There is already an important effort going on inside the Distributed Management Task Force (DMTF) [14] to develop standards that enable the interoperation of clouds. Also, different clouds can be combined by using *virtual infrastructure manager* such as Eucalyptus [15] or OpenNebula [16–18]. In a way, this resembles the evolution of grids, where new systems have been developed [19] to combine (i.e. federate [20]) the capabilities of different grid execution services.

To reach these goals, we advocate for the addition of a new abstraction layer on top of the infrastructure clouds' interfaces that allows us to control services as a whole, and frees SPs from the hurdles of manually controlling potentially large sets of virtualized resources. This would change the way that SPs interact with clouds. Instead of dealing with the management of hardware resources on different platforms (as it is shown in Fig. 1(a)), they will only need to use a single interface for service management (see Fig. 1(b)), which is closer to the concepts typically managed by SPs. This new abstraction layer should run on top of a *Virtual Infrastructure Manager* (VIM) that enables the utilization of several clouds with heterogeneous interfaces from different CPs. Here we present *Claudia*, our proposal implementation of such layer, born as part of the EU-funded RESERVOIR project [21]. Parts of its code will be released as open source (Affero GPL licensing) during project execution. Also, a proprietary extension of Claudia will be

[1] http://www.gogrid.com/downloads/GoGrid-scaling-your-internet-business.pdf.

[2] http://opennebula.org/.



(a) Today: SP manages VMs in different CPs.

(b) Our proposal: an unique interface to handle the service.

Fig. 1. Our proposal is based in the addition of a new layer that abstracts the management of different VMs running in different clouds, presenting instead an unique interface for the management of the *service*.

tested in production environment in Telefónica (in the context of Telefónica's Cloud efforts), where Claudia's code is being further refined. An important feature of Claudia is that it runs on top of a VIM that can, in turn, use different infrastructure providers. Hence, services are deployed in different clouds, which form a transparent federation.

2.1. Service definition format candidates

The first key gap to address to build service-aware clouds is the lack of standard formats that SPs can use to specify how their services must be deployed and operated by the cloud platform. Existing standards such as Solution Deployment Description (SDD) [22] and Configuration Description, Deployment, and Lifecycle Management (CDDLM) [23] could be pondered. However, the former is oriented to software distribution (development, maintenance, etc.), not covering runtime (e.g. deployment or reconfiguration/scaling). The latter is too abstract (an agreement should be reached by different parties to make it interpretable). Although it covers some deployment descriptions, it lacks features for pure hardware definition. A specification is needed that covers not only software distribution, but lifetime constrains as well (like, for instance, the way the service should be scaled). Also, this specification shall be concrete enough for including hardware constraints, while being generalizable to cover runtime stage aspects such as scalability or configuration dependencies among service components (e.g. the IP address of the database backend can be required by the web frontend) that can only be solved at deployment time. Open Virtual Format (OVF) [24] is a standard oriented to virtual applications description which we deem more fitted to service definition in Clouds, as it can address all the requirements involved thanks to its extensions mechanism that enables the addition of new capabilities. In [25] we discussed in more detail the features and limitations of each standard (SDD, CDDLM, etc.). In Section 3.1, we introduce the extensions we have added to OVF to make it suitable for service definition in cloud environments. This extended version of OVF is the format of choice that we use to define the services that will be controlled by Claudia (i.e., Claudia expects services to be described in that format).

3. Enabling service-level management in clouds

Claudia automates the deployment and scaling of services, providing them with an appropriate *Service Abstraction Level* (see previous section). Each service in Claudia is defined by its corresponding Service Description File (SDF, see Section 3.1).

Claudia uses a staged deployment mechanism that initiates components taking into account the dependencies expressed by the SP, so no component is started before all the other components it depends on are running. Furthermore, to make the service deployment a fully automated process, Claudia allows for *automatic components customization*. The SP can configure in the SDF the customization data required by each component that are not known before deployment (like the IP assigned to some other component). Claudia takes care of preparing the customization data and making them available to the components that need them using the OVF Environment mechanism described in [24]. Thus, Claudia is able to deploy whole services in a single step, without any manual intervention from the SP.

Claudia implementation of the *Automatic Scalability* requirement is based on the *scalability rules* defined by SPs for each service in the SDF. A scalability rule associates sets of conditions to sets of actions. Conditions can be based on user-defined service-level metrics rather than using hardware metrics only. Responsiveness at a service level calls for custom application-specific metrics to be seamlessly integrated in the system. Claudia continuously checks the arriving information (infrastructure and/or service metrics) against the user-defined scalability rules and, when the triggering conditions are met, it activates the consequent actions.

The automatic scalability functionality of Claudia is combined with *Smart Scaling* capabilities based on a business information model containing business rules (e.g. keep monthly cost below 500) which influence the service lifecycle. For instance, it may not be in the SP interest to increase the number of servers of a given type if the budget is exceeded. Nevertheless, the application could keep on running by redeploying some servers back from the external IaaS provider into the locally managed datacenter.

Finally, Claudia is designed to avoid *vendor lock-in* by using a VIM to allocate and release virtual resources such as VMs. By using VIMs with extended capabilities to interface with different clouds, the SP is able to achieve vendor independence. Claudia does not implement this functionality itself, as it is only concerned about the management of services, but can easily be "plugged" to any of these managers.

The OpenNebula Virtual Infrastructure Manager [16], thanks to its hybrid cloud computing functionality, can be used as the VIM. OpenNebula provides the functionality needed to deploy, monitor

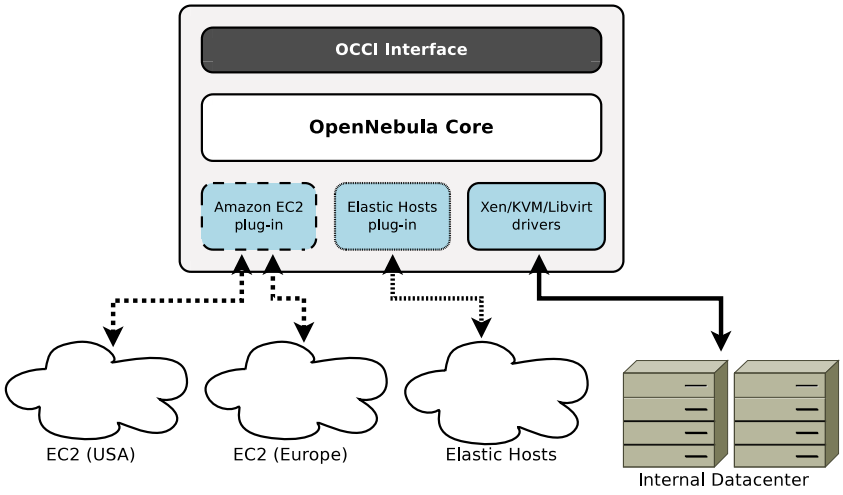


Fig. 2. OpenNebula as a virtual infrastructure manager with extended scaling out capabilities.

and control VMs on a pool of distributed physical resources, usually organized in a cluster-like architecture, but can additionally incorporate remote cloud providers. In particular the EC2 and Elastic Hosts (EH) adapters convert the general requests made by the virtualization components of the OpenNebula core to deploy, shutdown and monitor a VM using the EC2 and the EH APIs (see Fig. 2).

3.1. SDF syntax

Services in Claudia are defined by *Service Description Files* (SDF), whose syntax is based on DMTF's OVF standard [24]. OVF is a vendor and platform neutral portable packaging mechanism for Virtual Appliances (VA), i.e. pre-configured software stacks comprising one or more virtual machines to provide self-contained services. In cloud computing context, the VA is the service to be deployed by the SP in the cloud platform.

The OVF standard specifies the *OVF descriptor*. It is a XML file (conforming to several XML Schemas included as normative documents in the specification) enclosing information regarding the VA and its constituting Virtual Machines (VMs). It is composed of three main parts:

- A description of the files included in the VA, as <Reference> tags for virtual disks, ISO images, internationalization resources, etc.
- Meta-data about all virtual machines contained in the package. For example, <DiskSection> describes virtual disks data and <NetworkSection> provides meta-information regarding the logical network interconnecting the VMs in the VA.
- A description of the different VM systems, in <VirtualSystem> tags. OVF allows to specify the virtualization technology (e.g., "kvm" or "xen-3") and to describe the virtual hardware each VM needs (<VirtualHard-ware>) in terms of CPU, memory and hardware devices (including disks and network interfaces) to be provided by the underlying hypervisor running the VM.

  Additional information can be provided in the form of meta-data sections. For example, <ProductSection> specifies product information for software packages in VMs, including key-value properties using ovf:key and ovf:value attributes in <Property> tags.

  Finally, VMs can be grouped in collections (<VirtualSystemCollection>) which make sense to group common information regarding several VMs. Within the collection, the <StartupSection> defines the virtual machine booting sequence.

Note that both VA and VM meta-data descriptions are structured in sections (corresponding to <*Section> tags). One important feature of OVF is that it can be extended with new sections beyond the standard ones. Other possible extension mechanisms are the definitions of new elements in existing sections and/or new attributes in existing tags.OVF extensibility is an important property from the point of view of the work developed in this paper as we have applied it to adapt the standard OVF format to cloud environments in a non-disruptive way.

Apart from the OVF descriptor, the standard also describes a guest-host communication protocol, which is used by the deployment platform to provide configuration parameters that the guest could need at deployment time, e.g., networking parameters, service parameters, etc. This communication is based on a XML document (named *OVF Environment document*), built by the deployment platform, in which a list with the different properties defined in the <ProductSection> in the OVF descriptor is specified as key-value pairs. This file is passed to the VM within a dynamically generated CD/DVD ISO image in whose root directory is the XML file containing the OVF environment document. The file is read by the software running in the VM during the boot process, thus enabling the automatic configuration of application parameters.

Standard OVF as described in [24] was designed considering conventional IT infrastructure (i.e. data centers) as the target deployment platform, so it presents some limitations when applied directly to service deployment in clouds. Therefore, in order to make it a valid syntax for SDF files addressing the requirements defined in Section 2, several extensions have been developed to adapt OVF to cloud computing. They are summarized in the following subsections, more detail can be found in our previous work [25]. It is worth mentioning that part of these extensions are currently being proposed to the DMTF Cloud Incubator Working Group, to be discussed within the scope of the DMTF and eventually being standardized.

A complete SDF example corresponding to our first experiments in Section 4 can be found in Appendix.

3.1.1. Extensions to support automatic scalability

In order to avoid the limitations in the current scalability approaches in existing IaaS cloud offerings (see Section 2), two new OVF sections have been defined.

First, the <KPIsSection> enables the SP to define the relevant Key Performance Indicators (KPIs) which values govern the scalability of the service. For example, the scalability can be controlled by the queue size of a given process, so if the

queue grows beyond a given threshold, more resources need to be allocated. It is worth mentioning that the SP can freely define whichever KPI she wants, totally suited to the specific service being deployed, as far as the SP also provides the proper software (i.e. probes) installed in the VMs in order to report periodically and automatically the KPI value to the cloud platform. The <KPIsSection> just contains a list of <KPI> tags, each one with a single attribute KPIname to describe the name of a KPI.

Second, the <ElasticityRulesSection> describes the actual user-defined elasticity rules which, making use of the aforementioned KPIs, state how the service has to be scaled. Each rule contains a *condition* (or set of conditions) to be periodically supervised by the cloud infrastructure. When those conditions are met, the rule is triggered and an *action* or set of actions are executed. For example, *when the amount of tasks in the queue size per instance is greater than* 20 *(condition), then a new instance of VM of type "Executor" (action) must be created*.

The <ElasticityRulesSection> is structured in a set of one or more <Rule> tags, each one describing a scalability rule. Each rule in sequence is defined by three tags. The tag <Name> contains the rule name; the tag <Trigger> is used to define the conditions that when fulfilled will fire the rule and the frequency the rule must be checked; finally the tag <Action> describes the actions to perform when the rule is triggered. Within the condition part, the KPIs defined in <KPIsSection> can be referred, using the @kpis token. For example, if a KPI named QueueLength was defined in <KPIsSection>, then it is refereed within <Trigger> as @kpis.QueueLength. Particular examples of elasticity rules can be found in Section 4.

In addition, we have defined three new attributes for the <VirtualSystem> tag. In our approach, this tag is used to defined the different VM types that form the service (classes), instead of defining individual VM (instances) as in standard OVF. In order to control the elasticity bounds, min and max attributes specify respectively the minimum and maximum number of instances of the corresponding <VirtualSystem> that can be created. A third attribute, initial, defines the initial number of instances to be created at service deployment.

3.1.2. Extensions for deployment time customization

Using <ProductSection> and the OVF Environment mechanism described before, OVF supports the customization of services with parameters that are known by the user prior to deployment or that she can provide manually at deployment time. However, in a cloud environment it is required also to define certain features that are only known *at deployment time without the user interaction*. For example, IP addresses assigned to VMs cannot be known before deployment time, because they depend on a dynamic pool of available IPs whose status cannot be predicted. Hence they cannot be set manually by the SP, and they are managed internally by the cloud. However, IP addresses need to be passed to VMs as they are needed to configure network interfaces (when DHCP is not used) or to interconnect different VMs (e.g. front-end connections to the database).

In order to solve this issue, we have extended the property definition within <ProductSection> using certain *macros* for ovf:value. Each macro is identified by the 'at' symbol (@) as prefix and is translated to an effective value depending on the macro name. It is worth mentioning that this is transparent from the point of view of standard OVF implementations but recognized by the cloud deployment software supporting this extension. First, standard OVF authoring tools would deal with "@-prefixed values" without giving any special meaning and, secondly, an standard OVF Environment document will be generated and passed to the VM.

The following macros have been defined.

- To specify IP addresses: @IP([network_name]), which is replaced by the IP assigned by the cloud middleware to the virtual machine in *network_name*. The *network_name* has to match with one of the networks names, i.e. name attribute in some of the <Network> tags within the <NetworkSection>, to which the VM in which the macro is used is connected.
- To specify the instance number within the class of virtual systems: @VMid, which is resolved to the instance number. Considering the semantic in which <VirtualSystem> represents not individual instances but component types, the first instance will get @VMid 0, the next @VMid 1, and so on, etc.
- To refer to properties actually defined in sibling virtual systems within the same <VirtualSystemCollection>: @vs.prop, where the property key in the sibling system is referred by *prop* and the virtual system is identified by *vs*.
- To refer to the qualifier and channel used for the KPI monitoring, related with the KPI specification described in Section 3.1.1. In particular, the @KPIQualifier is used to specify the service label that has to be appended to the KPI name, in order to make KPI notification uniquely so avoiding conflict among KPIs with the same name from different services. The cloud platform ensures the @KPIQualifier value is different in each service deployed. In addition, given that the address (IP and port) of the endpoint or channel for KPIs value notification is not known by the SP, the @KPIChannel macro is used. When a new virtual machine is created, the cloud platform will assign the endpoint address to the properties that contain that macro, so the probes within the VM to monitor the KPIs can be configured properly.

3.1.3. Extensions for external connectivity specification

OVF enables us to define the networks to interconnect the different VMs conforming the service using the <Network> tag within the <NetworkSection>. The network model considers them just as interconnection points, and OVF lacks the ability to specify network properties. Complex services deployed in clouds uses two kind of networks: internal (e.g. the network to connect service front end and back end) and public (e.g. the network to connect the front end to service users through the Internet). However, the two different types of networks cannot be expressed in OVF but the deployer needs to know that information in order to properly configure the access and security levels associated with each network, e.g. VMs could get public addresses in the external networks and private ones in the internal networks.

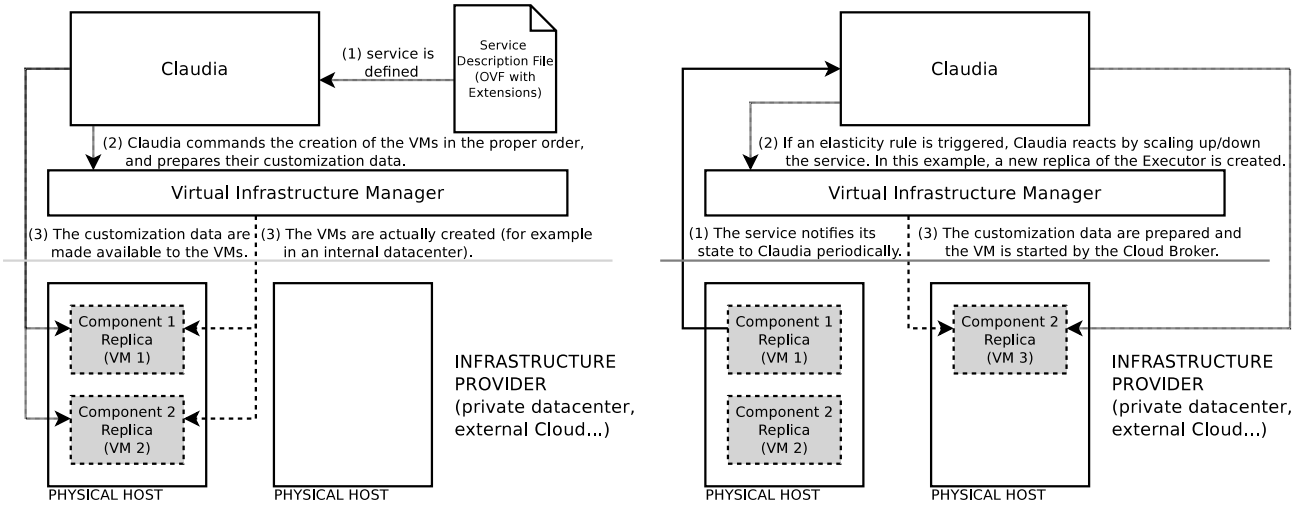
In order to overcome this limitation, we have defined the public attribute for <Network> tags to mark if a network is public (public = "yes") or not (public = "no") or omitted (implying a "no").

4. Experimental evaluation: three different scalability mechanisms

In order to illustrate the above features, we have prepared a deployment of a Grid service based on the Sun Grid Engine (SGE) [26] platform, using OpenNebula as the VIM. A SGE deployment consists of a *master* node that controls one or more *executor* nodes, which perform computation tasks on a set of data. The master node handles the queue of tasks to be run, balances the load among the executors and receives back the processed data from them.

4.1. Automatic customization of SGE components

There is a dependency among the SGE components that needs to be solved at deployment time. When an executor is started, it registers itself on the master so the master knows that a new executor is ready to receive jobs. Hence, all executors need to



(a) Service automatic deployment.

(b) Service automatic scalability.

Fig. 3. Claudia automates different parts of the service lifecycle like the initial deployment of the whole service and the scaling of resources depending on the service status, given by user-defined, service-level metrics.

```
<Property ovf:key="masterIP" ovf:type="string" ovf:value="@Master.IPforMaster"/>
```

Fig. 4. Specification of the dynamic configuration of the SGE executors (master's IP).

```
<rsrvr:Rule>
    <rsrvr:Name>ScaleUp</rsrvr:RuleName>
    <rsrvr:Trigger condition=
           "@{kpis.QueueLength}/@{components.Executor.replicas.amount} &gt; 50"/>
    <rsrvr:Action run="createReplica(components.Executor)"/>
</rsrvr:Rule>
<rsrvr:Rule>
    <rsrvr:Name>ScaleDown</rsrvr:RuleName>
    <rsrvr:Trigger condition=
           "@{kpis.QueueLength}/@{components.Executor.replicas.amount} &lt; 20"/>
    <rsrvr:Action run="removeReplica(components.Executor)"/>
</rsrvr:Rule>
```

Fig. 5. Scaling up/down rules to adapt the SGE service to the demand.

know the IP address of the master. Nonetheless, this datum is not known before the master is running (and so the deployment has been started). To manage this, the SP uses the SDF to configure the deployment process so Claudia starts the master first. In the same manner, the SDF commands Claudia to make the IP address of the master available to all executor replicas (the datum is passed using the OVF environment mechanism described in Section 3.1). Once these dependencies are set in the SDF document, Claudia automatically deploys the SGE service without any intervention from the SP (see Fig. 3(a)).

These dependencies are indicated in our SDF by means of special-purpose OVF <Property>s (see Section 3.1.2). In the case of getting the `masterIP` known to the executors, the OVF code would look as shown in Fig. 4.

4.2. Automatic scalability based on the SGE status

A key feature to be controlled during the lifecycle of the service is the amount of resources allocated for it. To avoid both potential shortages or overprovisioning of resources the SP must be able to define how Claudia must assign resources depending on the service status (load). In this case, let us assume that the SGE SP knows that the performance greatly decreases when the ratio of tasks per executor is greater than 50. As described in Section 3.1.1, an scalability rule can be expressed in the SDF to define the scale up actions that ensure this service feature is being considered.

These rules (as specified in the OVF-based SDF) can be used for several scaling purposes. As we will demonstrate in the following subsections, Claudia allows for several types of scaling mechanisms.

- *Service component scale up/down*: including new service component replicas and balancing load among them.
- *Service component reconfiguration*: adding more/less powerful service components (adding bigger/smaller VMs for the SGE executor nodes, in our case study).
- *Service component scale in/out*: deploying part of the service components in another Cloud provider by means of the infrastructure management layer. That is, the service is deployed in a *federated cloud environment* with components hosted in different clouds, in a transparent manner.

4.2.1. Service component scale up/down

Fig. 5 shows a service component scaling up rule, where we can observe how Claudia allows the SP to define the scalability actions depending on the *service status* (the queue size in this case). A similar rule could be used for scaling down the service, in case the SP wants to prevent the overprovisioning of resources, for example by commanding to remove an idle executor replica when the ratio of tasks per executor gets lower than a certain threshold. Fig. 3(b) shows the interactions involved in the scaling of a service.

In Fig. 6, we show how the system is scaled following the rules defined by the SP. We can observe how when a burst of jobs arrives to the master the queue length rapidly increases. The `ScaleUp` rule is then triggered and after some time a new executor replica appears (this lag is due to the time consumed in booting the new VM). As soon as the new executor replica starts processing tasks,

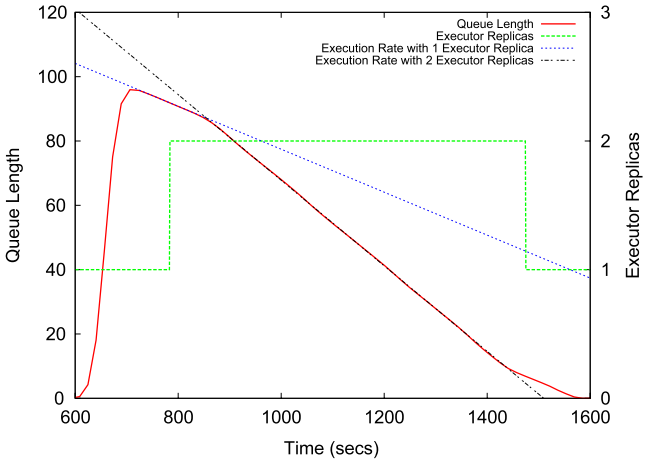


Fig. 6. Claudia enables service scaling according to the current load.

the queue length decreases at a faster pace (note the different slopes). Later, when the queue length is lower than a certain threshold (less than 10 tasks per executor replica), the `ScaleDown` rule is triggered to remove one executor replica. This replica is still present while it runs its internal shutdown process, although it does not run any more jobs. This explains why the execution rate changes before the replica is actually removed.

Besides, in Claudia the SP can also set upper or lower bounds to the amount of resources created, so for example the SGE service never has less than one nor more than two executors. This is specified in the `<VirtualSystem>` tag within the `<VirtualSystemCollection>` in the OVF-enabled SDF (see Fig. 7). The upper limit helps the SP to control the costs of the service, and the lower limit serves to make sure that there are always some resources ready to process new jobs.

4.2.2. Service reconfiguration

In the previous section, Claudia demonstrated its powerful mechanism for scaling up/down a given service by adding/removing service instances as needed. This was easily done by specifying simple high level rules that are closer to the user mindset than traditionally used infrastructure level rules and metrics. These simple scaling rules can be packaged together with more sophisticated strategies aiming to reduce the price, optimize the reached performance or both. Thus, more sophisticated strategies can also be defined by using Claudia. Claudia allows for the addition of any action treating the service as a whole entity using service level metrics, further expanding the current state of the art scaling technologies in Cloud systems.

In order to exemplify this, in this subsection we present a use case in which mid sized and big sized images are available that help SPs modulate the obtained performance/cost of the deployed service. The SP sets reconfiguration rules in the SDF, indicating that upon the occurrence of some conditions (number of normal executors bigger than 2, see Fig. 8), the current number of executor replicas should be replaced by a big sized executor. Thus, three normally sized replicas are removed and a big sized replica is created.

The rule in Fig. 8 exemplifies the runtime reconfiguration potential capabilities allowed by Claudia. Reconfiguration rules are often used in synchrony with simpler scaling rules (as shown in the previous section and in Fig. 6). Thus, the employed set of rules for this use case contains rules for scaling up/down normal executors, for scaling up/down big executors, and rules for reconfiguring the service (the one shown in Fig. 8). This set of rules can be described as follows (and formally defined as in Fig. 9):

First rule. Scaling up a normal executor. *Condition*: number of tasks per executor node greater than 50 and less than 3 normal executors running. *Action*: Create a new normal executor.

Second rule. Scaling up a big executor (and removing a normal one). *Condition*: number of tasks per executor node greater than 50 and 3 or more normal executors running. *Action*: replace a normal executor with a big one.

Third rule. Scaling down a normal executor. *Condition*: number of tasks per executor node lower than 20, there are no big executors and at least a normal executor running. *Action*: remove a running normal executor.

Fourth rule. Scaling down a big executor (and replacing it with a normal one). *Condition*: number of tasks per executor node lower than 20 and at least a big executor running. *Action*: replace a big executor with a normal one.

Fifth rule. Reconfiguration rule. *Condition*: number of normal executors is 3 or more. *Action*: remove 3 normal executors and replace them with a big one.

Fig. 10 shows the results of a sudden increase in the number of jobs in the queue of the SGE master. As load increases so does the number of normal executors running by virtue of `ScaleUp` enforcement. Then (right after second 1000), since three normal executors are running, by `Replacement`, these should be removed and a big one should be created by replacing them. However, a minimal number normal executors, one, was set in the SDF, so one is always left. Later (around second 1300), the number of big executors is again increased due to `ScaleUpBig` application. Two new normal executor replicas are added (making up a total of three normal replicas) due to `ScaleUp` enforcement. Right before the creation of the third normal executor replica, `ScaleUpBig` application results in a new big executor (three in total). Again, since three normal executors are available, `Replacement` removes them (only two, since the SDF-indicated minimum is 1) and tries to replace them with another big one, which cannot be done since the maximum allowed number of big executors is 3, as stated in the SDF. Around second 2000, `ScaleUp` is triggered again, and a new normal executor is created. This steps leads to a "steady state" (two normal executors and a big one) in which the system is among threshold values of the rules and performs without changes. As load decreases (around second 3300), `ScaleDownBig` is triggered twice, taking the number of normal executors to 4 and the number of big executors to 1. Being the number of normal executors 4 (>3), `Replacement` is enforced

```
<VirtualSystem ovf:id="Executor" rsrvr:min="1" rsrvr:max="2" rsrvr:initial="1">
```

Fig. 7. Capping of the maximum number of service replicas for a service component.

```
<rsrvr:Rule>
    <rsrvr:Name>Resizing</rsrvr:RuleName>
    <rsrvr:Trigger condition="@{components.Executor.replicas.amount} &gt; 2"/>
    <rsrvr:Action run="createReplica(components.BigExecutor);
                       removeReplica(components.Executor,3);"/>
</rsrvr:Rule>
```

Fig. 8. Resizing of service components to adapt to variable demand.

```
<rsrvr:Rule>
    <rsrvr:Name>ScaleUp</rsrvr:RuleName>
    <rsrvr:Trigger condition=
           "(@{kpis.QueueLength}/(@{components.BigExecutor.replicas.amount}*3 +
           @{components.Executor.replicas.amount} +1)&gt; 50) &amp;&amp;
           (@{components.Executor.replicas.amount} &lt; 3)"/>
    <rsrvr:Action run="createReplica(components.Executor)"/>
</rsrvr:Rule>
<rsrvr:Rule>
    <rsrvr:Name>ScaleUpBig</rsrvr:RuleName>
    <rsrvr:Trigger condition=
           "(@{kpis.QueueLength}/(@{components.BigExecutor.replicas.amount}*3 +
           @{components.Executor.replicas.amount} +1)&gt; 50) &amp;&amp;
           (@{components.Executor.replicas.amount} &gt; 2)"/>
    <rsrvr:Action run="createReplica(components.BigExecutor);
                       removeReplica(components.Executor);"/>
</rsrvr:Rule>
<rsrvr:Rule>
    <rsrvr:Name>ScaleDown</rsrvr:RuleName>
    <rsrvr:Trigger condition=
           "(@{kpis.QueueLength}/(@{components.BigExecutor.replicas.amount}*3 +
           @{components.Executor.replicas.amount} +1)&lt; 20) &amp;&amp;
           (@{components.BigExecutor.replicas.amount} == 0) &amp;&amp;
           (@{components.Executor.replicas.amount} &gt; 0)"/>
    <rsrvr:Action run="removeReplica(components.Executor)"/>
</rsrvr:Rule>
<rsrvr:Rule>
    <rsrvr:Name>ScaleDownBig</rsrvr:RuleName>
    <rsrvr:Trigger condition=
           "(@{kpis.QueueLength}/(@{components.BigExecutor.replicas.amount}*3 +
           @{components.Executor.replicas.amount} +1)&lt; 20) &amp;&amp;
           (@{components.BigExecutor.replicas.amount} &gt; 0)"/>
    <rsrvr:Action run="createReplica(components.Executor,1);
                       removeReplica(components.BigExecutor);"/>
</rsrvr:Rule>
<rsrvr:Rule>
    <rsrvr:Name>Replacement</rsrvr:RuleName>
    <rsrvr:Trigger condition=
          "@{components.Executor.replicas.amount} &gt; 2"/>
    <rsrvr:Action run="createReplica(components.BigExecutor);
                       removeReplica(components.Executor,3);"/>
</rsrvr:Rule>
```

Fig. 9. Reconfiguration of service components to adapt to a user-defined performance/cost strategy.

by creating a big executor and removing three normal executors leading to two big and one normal (at second 4000). Then again, load per executor decreases and `ScaleDownBig` application results in two consecutive creations of normal executors and removal of big executors (around second 4200). Having no big executors and three normal ones left, `ScaleUpBig` enforcement leads to the removal of a normal executor. Immediately afterwards, `ScaleUpBig` action is triggered leading to a status with a big executor and a normal executor (around second 4400). At second 4700, `ScaleDownBig` comes into scene followed altogether by the application of `ScaleDown`, driving the system to a final state of a single normal executor running for minimal (or null) workload.

Making a mechanical or electrical engineering analogy, we could explain the observed behavior by comparing the transitions to oscillations in an harmonic oscillator. In this setting, the system behaves like an underdamped system (damping ratio <1) in those phases close to rule-defined thresholds. The system oscillates with the "oscillation amplitude" gradually decreasing to zero. An underdamped response is one that oscillates within a decaying envelope. For instance, from no executors the system "oscillates" to a maximum of 3 ($\Delta = 3$), in the next cycle the "oscillation" leads to a $\Delta = 2$ (one normal executor back to three and again back to one). In the final oscillation before steady state, the system increases in one executor ($\Delta = 1$). A similar oscillating pattern is observed after the steady state. The same "underdamped" behavior applies for the number of big executors, although the dampening ratio is equal to 0 for the observed period.

Also, a smoother response can be appreciated when compared to Fig. 11. In that experiment, the same scalability rules used for Fig. 10 are applied; however, we can see how in several occasions two normal executors are removed but no new big executor is created as it should. This is because different bounds to the amount of

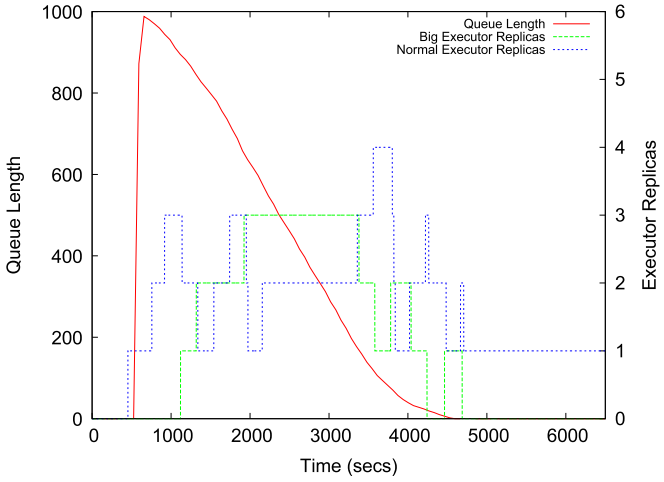


Fig. 10. Claudia enables user-defined service reconfiguration together with scaling policies.

big executors were set for this experiment: not more than one big executor could be created. The goal of this experiment is to show a second scalability mechanism that can easily be implemented by Claudia. An important lesson learned was the strong relation found between scaling rules and components bounds.

4.2.3. Service component scale in/out (transparent cloud federation)

For this next use case, let us now imagine a scenario in which the SP would prefer not to install all the replicas of a certain component in only one cloud but in two (or more). This could be a useful

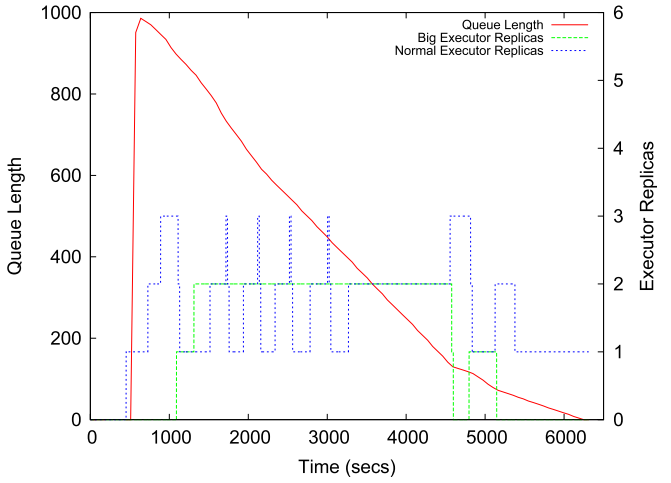


Fig. 11. Claudia enables user-defined service reconfiguration together with scaling policies. Attention should be paid to replica boundaries.

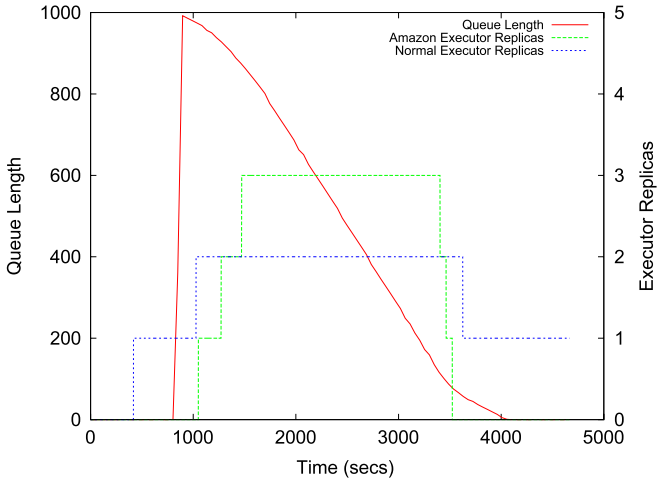


Fig. 13. Claudia can deploy components in different Clouds for the same service.

scenario in the case of having availability concerns (if one cloud crashes there still be replicas in other cloud(s) to keep running the service). Also, economical reasons can be applied here: if the overall expenditure in a cloud hugely increases, then put as many service components as possible in the cheaper one. Yet another reason to deploy components in an external cloud can be a shortage of resources in the local facilities.

This is achieved through *scale in/out* actions that allow us to deploy all or part of the service to a more suitable CP. Upon an increase in demand, part of the service is moved to a more convenient CP. Here we provide a use case illustrating Claudia's capabilities (see Fig. 12).

First rule. Scaling up one normal executor. *Condition*: number of tasks per executor node greater than 50 and there are less than two normal executors running. *Action*: create new local executor replica.

Second rule. Scaling out one Amazon executor. *Condition*: number of tasks per executor node greater than 50 and there are three or more normal executors running. *Action*: create new executor in Amazon.

Third rule. Scaling down one normal executor. *Condition*: number of tasks per executor node lower than 20 and there are no Amazon executors. *Action*: remove normal (local) executor.

Fourth rule. Scaling in one Amazon executor *Condition*: number of tasks per executor node lower than 20 and there is at least one Amazon executor running. *Action*: remove Amazon executor.

Fig. 13 shows the evolution of the SGE service in this federated setting. A starting replica is created, given that the client set the initial normal (local) executors to 1. Upon reception of a sudden increase in the number of incoming jobs, the `ScaleUp` condition is met and another normal executor replica is created (a total of two at second 1000). Since the optimal load per executor (as so considered by the SP) has not been reached, the system keeps on scaling. But this time `ScaleOut` comes into play and three new Amazon executor replicas are created (the maximum number for

```
<rsrvr:Rule>
    <rsrvr:RuleName>ScaleUp</rsrvr:RuleName>
    <rsrvr:Trigger condition="
           (@{kpis.QueueLength}/(@{components.AmazonExecutor.replicas.amount}+
           @{components.Executor.replicas.amount}+1)&gt; 50) &amp;&amp;
           (@{components.Executor.replicas.amount} &lt; 2)"/>
    <rsrvr:Action run="createReplica(components.Executor)"/>
</rsrvr:Rule>
<rsrvr:Rule>
    <rsrvr:RuleName>ScaleOut</rsrvr:RuleName>
    <rsrvr:Trigger condition="
           (@{kpis.QueueLength}/(@{components.AmazonExecutor.replicas.amount}+
           @{components.Executor.replicas.amount}+1)&gt; 50) &amp;&amp;
           (@{components.Executor.replicas.amount} &gt; 1)"/>
    <rsrvr:Action run="createReplica(components.AmazonExecutor)"/>
</rsrvr:Rule>
<rsrvr:Rule>
    <rsrvr:RuleName>ScaleDown</rsrvr:RuleName>
    <rsrvr:Trigger condition="
           (@{kpis.QueueLength}/(@{components.AmazonExecutor.replicas.amount}+
           @{components.Executor.replicas.amount}+1)&lt; 20) &amp;&amp;
           (@{components.AmazonExecutor.replicas.amount} == 0)"/>
    <rsrvr:Action run="removeReplica(components.Executor)"/>
</rsrvr:Rule>
<rsrvr:Rule>
    <rsrvr:RuleName>ScaleIn</rsrvr:RuleName>
    <rsrvr:Trigger condition="
           (@{kpis.QueueLength}/(@{components.AmazonExecutor.replicas.amount}+
           @{components.Executor.replicas.amount}+1)&lt; 20) &amp;&amp;
           (@{components.AmazonExecutor.replicas.amount} &gt; 0)"/>
    <rsrvr:Action run="removeReplica(components.AmazonExecutor)"/>
</rsrvr:Rule>
```

Fig. 12. Scaling out of service components to adapt to variable demand.

normal replicas is kept to 2 and the maximum for Amazon replicas is kept to 3). In general, the system behavior is defined by the set of rules and boundary conditions (max and min values). Setting these determines the oscillating pattern. This time, it resulted to be an underdampened-like system with the dampening ratio equal to 0.

We should remark that *this experiment has tremendous implications for cloud federation*. Claudia is enabling SPs to seamlessly deploy their services across different CPs (thus mitigating the vendor lock-in problem) and helping them to decide and automate the number of service replicas in every CP according to the local site load, to the service needs, changes in prices or variable budgets, conditions that certainly occur on a daily basis in current business environments.

4.3. Smart scaling by business logic control

Scaling for the sake of performance reasons is an important step forward to deliver the "unlimited resources" vision promised by cloud technologies. With Claudia SPs provide the SDF file describing the initial requirements of the service, along with the scalability rules and the maximum and minimum resources that can be allocated for the service. However, having enterprises on the cloud still faces additional challenges that cannot be solved from a "service performance only" perspective.

The provider of the infrastructure may want to set its own control on the resources allocated per SP, depending on the SP client record (i.e. is this SP a good client?) and/or business preferences (e.g. limiting the number of replicas due to an excessive cost). This custom business strategy is needed to cap or widen the initial deployment and the automatic scaling of the service as needed. The Business Logic Analyzer (see Fig. 14) is the component of Claudia that sets the real deployment boundaries, which can differ from the ones set by the SP, to not exceed, for instance, a maximum allowed cost. Some flexibility is allowed here when, for example, a "good client" asks for a deployment exceeding his current budget, the Business Logic Analyzer lets the deployment continue until a configurable maximum negative balance. Specific business strategies require specific metrics to allow for the appropriate expressivity level.

In this set of experiments, a good client was given extra trust so that she could perform a deployment in the absence of enough credits. Claudia automatically recognized the *importance* of the client by using a set of business rules stored in a business information model. This way, Claudia brings an additional criteria for the CP to control the deployment environment. A normal client with no credit record is not given these trusties and, thus, unable to deploy without enough credits in the system.

5. Challenges on the horizon

The SGE use cases show how Claudia evolves present cloud solutions to provide a flexible environment where (1) an appropriate abstraction level is offered to the SP for defining her services in a SDF document; (2) whole services are deployed with no SP intervention, as components dependencies and configuration are already defined in the SDF; (3) the service is automatically scaled up/down, reconfigured, or in/out, seamlessly federating sparse and heterogeneous resources, according to user-defined rules that include custom service metrics; (4) the locally managed datacenter and/or any external CP can be used to outsource resources in order to optimize the performance/cost ratio according to a custom business strategy and to minimize the impact of future vendor lock-in; (5) business rules play a key role in controlling service scalability.

Although Claudia opens some breaks, there are still cloudy issues along the way.

5.1. The chase for flexibility goes on

When defining the scalability rules of the service, the action to run when the rule is triggered is chosen by the SP from a certain set. At this moment, this set is still limited. New actions are being added such as for instance, the capability to trigger alarms aimed at an operator panel.

The Claudia team is also working on providing the mechanisms to enable user-defined actions when a rule is not fulfilled. Claudia already solves one of the major issues: the inclusion of application-level user-defined metrics to be checked against when deciding on launching an action. However, less declarative methods and the establishment of a more complex grammar and an action dictionary to combine the actions triggered when a rule condition has been met would enable us to build custom and semantically richer actions.

In the same manner, the flexibility and automation of the business strategies supported by Claudia are not enough. For instance, the CP can only specify budget limits and she has to manually include the "goodness" of her client (the SPs).

5.2. Is there more to performance/cost optimization than flexibility?

Whatever the achieved level of flexibility, there are other factors that greatly affect the obtained performance/cost ratio. In this setting, load prediction techniques may prove good for enhancing responsiveness; e.g. before the load hits the system, we have an extra replica up and running. Artificial intelligence approaches where the system learns and adapts on-the-fly are useful here. Also, some of them may be useful for extracting meaningful rules [27] for the users not to miss the valuable feedback that automatically learned rules provide.

It is, thus, important to be well acquainted with the features of the service to be deployed in order to build profitable rules. For example, finding appropriate rules and studying traffic patterns of a number-crunching service may not be useful at all for, let us say, a financial market access service.

6. Conclusions

The fast growth of cloud computing as a key new paradigm in the IT landscape cannot be denied. However, despite its early success, cloud solutions carry some limitations due to the lack of maturity typical of new technologies. A key drawback of present IaaS offers is their orientation to the delivery of "plain" infrastructure, while SPs demand new, more friendly interfaces able to deal not just with individual resources, but with compositions of them (i.e. services).

The set of requirements and features introduced and implemented here enable the holistic management of the lifecycle of services on a federated cloud environment. Claudia tackles the required features; our results show how new cloud systems can close the gap between the SP needs and the functionalities offered: single deployment operation, automatic, smart (business-controlled) and diverse scalability, and addressing the issue of the vendor lock-in by providing seamless access to the resources of different CPs for a single service.

Acknowledgements

This work is partially supported by the EU-FP7 RESERVOIR project under Grant #215605. We thank all RESERVOIR members for their valuable input.

Appendix. SDF example

```
<?xml version="1.0" encoding="UTF-8"?>

<Envelope xmlns:ovf="http://schemas.dmtf.org/ovf/envelope/1"
xmlns="http://schemas.dmtf.org/ovf/envelope/1"
xmlns:rsrvr="http://reservoir.fp7.eu/ovf-extensions"
xmlns:vssd="http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/CIM_VirtualSystemSettingData"
xmlns:rasd="http://schemas.dmtf.org/wbem/wscim/1/cim-schema/2/CIM_ResourceAllocationSettingData"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://reservoir.fp7.eu/ovf-extensions reservoir_manifest.xsd">

    <References>
        <!-- Disk images used by the service -->
        <File ovf:id="master"
              ovf:href="http://cetus.dacya.ucm.es:5555/Images/hardy32_master.img"/>
        <File ovf:id="execution"
              ovf:href="http://cetus.dacya.ucm.es:5555/Images/hardy32_executor.img"/>
    </References>

    <DiskSection>
        <Info>Describes the set of virtual disks used by VMs</Info>
        <Disk ovf:diskId="master" ovf:fileRef="master" ovf:capacity="100GB"
              ovf:format="http://www.vmware.com/specifications/vmdk.html#compressed"/>
        <Disk ovf:diskId="execution" ovf:fileRef="execution" ovf:capacity="100GB"
              ovf:format="http://www.vmware.com/specifications/vmdk.html#compressed"/>
    </DiskSection>

    <NetworkSection>
    <!-- We assume a very simple topology, using just one network to which both
     master and executors are connected -->
        <Info>List of logical networks used in the service</Info>
        <Network ovf:name="private">
            <Description>Network to connect service components</Description>
        </Network>
        <Network ovf:name="public">
            <Description>Public Internet</Description>
        </Network>
    </NetworkSection>

    <rsrvr:KPIsSection>
        <!-- KPIs that will be reported by the service, that can be used
             in the scalability rules section -->
        <ovf:Info>String</ovf:Info>
        <rsrvr:KPIs>
            <rsrvr:KPI KPIname="QueueLength"/>
        </rsrvr:KPIs>
    </rsrvr:KPIsSection>

    <rsrvr:ElasticityRulesSection>
        <!-- Replica scaling up rule -->
        <rsrvr:Rule>
            <rsrvr:RuleName>ScaleUp</rsrvr:RuleName>
            <rsrvr:Trigger condition=
                   "@{kpis.QueueLength}/(@{components.Executor.replicas.amount}+1)&gt;50"/>
            <rsrvr:Action run="createReplica(components.Executor)"/>
        </rsrvr:Rule>
        <!-- Replica scaling down rule -->
        <rsrvr:Rule>
            <rsrvr:RuleName>ScaleDown</rsrvr:RuleName>
            <rsrvr:Trigger condition=
                   "@{kpis.QueueLength}/(@{components.Executor.replicas.amount}+1)&lt; 20"/>
            <rsrvr:Action run="removeReplica(components.Executor)"/>
        </rsrvr:Rule>
    </rsrvr:ElasticityRulesSection>
```

```
<!-- VM types description -->
<VirtualSystemCollection ovf:id="SunGridEngine">

    <StartupSection>
        <!-- Starting (and stopping) sequence for the service -->
        <Item ovf:id="Master" ovf:order="0" ovf:waitingForGuest="true" />
        <Item ovf:id="Executor" ovf:order="1" />
    </StartupSection>

    <!-- Master daemon configuration parameters -->
    <ProductSection ovf:class="com.sun.master" ovf:instance="Master">
        <Info>Product customization for the installed master software</Info>
        <Product>SGE Master</Product>
        <Version>1.0</Version>
        <!-- This is solved to the IP of the master in the public network,
             so later can be referenced in the executor configuration -->
        <Property ovf:type="string" ovf:key="IPforMaster" ovf:value="@IP[public]"/>
        <!-- End point to send the monitoring info (KPI values) to -->
        <Property ovf:type="string" ovf:key="KPIListener" ovf:value="@KPIMonitorEndPoint"/>
        <!-- Prefix to add to KPIs so Claudia knows to which service belong to -->
        <Property ovf:type="string" ovf:key="KPIQualifier" ovf:value="@KPIQualifier"/>
    </ProductSection>

    <!-- Executor daemon on the executor image configuration parameters -->
    <ProductSection ovf:class="com.sun.executor" ovf:instance="Executor">
        <Info>Product customization for the installed executor software</Info>
        <Product>SGE Executor</Product>
        <Version>1.0</Version>
        <!-- Reference by ovf:key to the IPforMaster property. -->
        <Property ovf:key="masterIP" ovf:type="string" ovf:value="@Master.IPforMaster"/>
        <Property ovf:type="string" ovf:key="KPIListener" ovf:value="@KPIMonitorEndPoint"/>
        <!-- The hostname must be configurable as it there will be different
             replicas of this VM -->
        <Property ovf:type="string" ovf:key="hostname" ovf:value="executor0@id"/>
        <Property ovf:type="string" ovf:key="KPIQualifier" ovf:value="@KPIQualifier"/>
    </ProductSection>

    <!-- Hardware configuration of VMs -->
    <VirtualSystem ovf:id="Master" rsrvr:min="1" rsrvr:max="1" rsrvr:initial="1">
       <Info>Master VM description</Info>

       <VirtualHardwareSection>
           <Info>Virtual Hardware Requirements: 512Mb, 2 CPU, 1 disk, 1 nic</Info>
           <System>
               <vssd:ElementName>Virtual Hardware Family</vssd:ElementName>
               <vssd:InstanceID>0</vssd:InstanceID>
               <vssd:VirtualSystemType>vmx-4</vssd:VirtualSystemType>
           </System>
           <Item>
               <rasd:Description>Number of virtual CPUs</rasd:Description>
               <rasd:ElementName>2 virtual CPU</rasd:ElementName>
               <rasd:InstanceID>1</rasd:InstanceID>
               <rasd:ResourceType>3</rasd:ResourceType>
               <rasd:VirtualQuantity>2</rasd:VirtualQuantity>
           </Item>
           <Item>
               <rasd:AllocationUnits>MegaBytes</rasd:AllocationUnits>
               <rasd:Description>Memory Size</rasd:Description>
               <rasd:ElementName>256 MB of memory</rasd:ElementName>
               <rasd:InstanceID>2</rasd:InstanceID>
               <rasd:ResourceType>4</rasd:ResourceType>
               <rasd:VirtualQuantity>256</rasd:VirtualQuantity>
           </Item>
           <Item>
               <rasd:AutomaticAllocation>true</rasd:AutomaticAllocation>
```

```
                <rasd:Connection>private</rasd:Connection>
                <rasd:ElementName>Ethernet adapter on service_network</rasd:ElementName>
                <rasd:InstanceID>3</rasd:InstanceID>
                <rasd:ResourceType>10</rasd:ResourceType>
            </Item>
            <Item>
                <rasd:AutomaticAllocation>true</rasd:AutomaticAllocation>
                <rasd:Connection>public</rasd:Connection>
                <rasd:ElementName>Ethernet adapter on public network</rasd:ElementName>
                <rasd:InstanceID>6</rasd:InstanceID>
                <rasd:ResourceType>10</rasd:ResourceType>
            </Item>
            <Item>
                <rasd:ElementName>Harddisk 1</rasd:ElementName>
                <rasd:HostResource>ovf://disk/master</rasd:HostResource>
                <rasd:InstanceID>5</rasd:InstanceID>
                <rasd:Parent>4</rasd:Parent>
                <rasd:ResourceType>17</rasd:ResourceType>
            </Item>
        </VirtualHardwareSection>
    </VirtualSystem>

    <VirtualSystem ovf:id="Executor" rsrvr:min="1" rsrvr:max="2" rsrvr:initial="2">
        <Info>Executor VM description</Info>

        <VirtualHardwareSection>
            <Info>Virtual Hardware Requirements: 256Mb, 1 CPU, 1 disk, 1 nic</Info>
            <System>
                <vssd:ElementName>Virtual Hardware Family</vssd:ElementName>
                <vssd:InstanceID>0</vssd:InstanceID>
                <vssd:VirtualSystemType>vmx-4</vssd:VirtualSystemType>
            </System>
            <Item>
                <rasd:Description>Number of virtual CPUs</rasd:Description>
                <rasd:ElementName>1 virtual CPU</rasd:ElementName>
                <rasd:InstanceID>1</rasd:InstanceID>
                <rasd:ResourceType>3</rasd:ResourceType>
                <rasd:VirtualQuantity>1</rasd:VirtualQuantity>
            </Item>
            <Item>
                <rasd:AllocationUnits>MegaBytes</rasd:AllocationUnits>
                <rasd:Description>Memory Size</rasd:Description>
                <rasd:ElementName>256 MB of memory</rasd:ElementName>
                <rasd:InstanceID>2</rasd:InstanceID>
                <rasd:ResourceType>4</rasd:ResourceType>
                <rasd:VirtualQuantity>256</rasd:VirtualQuantity>
            </Item>
            <Item>
                <rasd:AutomaticAllocation>true</rasd:AutomaticAllocation>
                <rasd:Connection>private</rasd:Connection>
                <rasd:ElementName>Ethernet adapter on service_network</rasd:ElementName>
                <rasd:InstanceID>3</rasd:InstanceID>
                <rasd:ResourceType>10</rasd:ResourceType>
            </Item>
            <Item>
                <rasd:ElementName>Harddisk 1</rasd:ElementName>
                <rasd:HostResource>ovf://disk/execution</rasd:HostResource>
                <rasd:InstanceID>5</rasd:InstanceID>
                <rasd:Parent>4</rasd:Parent>
                <rasd:ResourceType>17</rasd:ResourceType>
            </Item>
        </VirtualHardwareSection>
    </VirtualSystem>
</VirtualSystemCollection>

</Envelope>
```

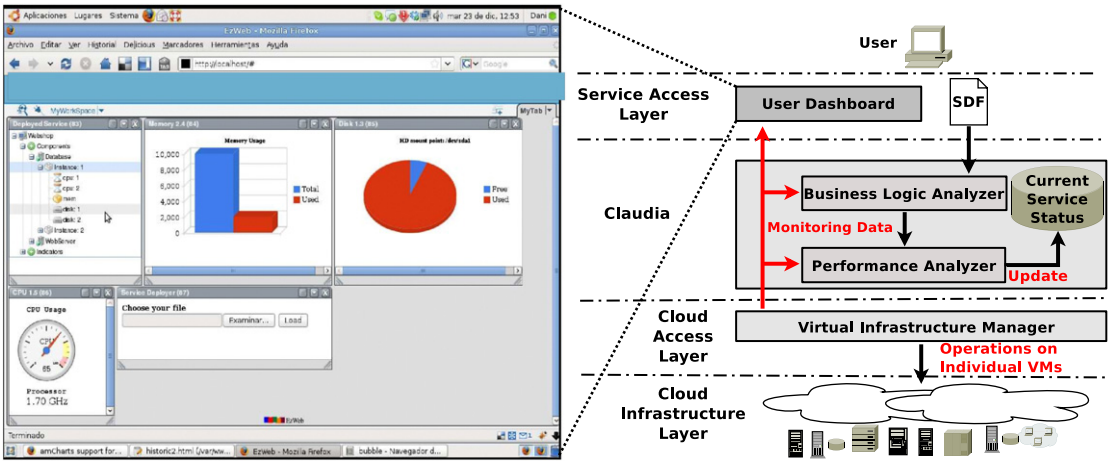


Fig. 14. Claudia brings an abstraction layer between the CP and the SP.

References

[1] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging IT platforms: vision, hype and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (6) (2009) 599–616.

[2] B. Hayes, Cloud computing, Communications of the ACM 51 (2008) 9–11.

[3] L. Vaquero, L. Rodero-Merino, J. Caceres, M. Lindner, A break in the clouds: towards a cloud definition, ACM Computer Communication Reviews 39 (1) (2009) 50–55.

[4] R. Buyya, C.S. Yeo, S. Venugopal, J. Broberg, I. Brandic, Cloud computing and emerging it platforms: vision, hype, and reality for delivering computing as the 5th utility, Future Generation Computer Systems 25 (6) (2009) 599–616. http://dx.doi.org/10.1016/j.future.2008.12.001.

[5] R.L. Grossman, Y. Gu, M. Sabala, W. Zhang, Compute and storage clouds using wide area high performance networks, Future Generation Computer Systems 25 (2) (2009) 179–183. http://dx.doi.org/10.1016/j.future.2008.07.009.

[6] J. Varia, Amazon white paper on cloud architectures. http://aws.typepad.com/aws/2008/07/white-paper-on.html (Sept. 2008).

[7] GoGrid web site: http://www.gogrid.com (Dec. 2009).

[8] The sun cloud API. http://kenai.com/projects/suncloudapis (Dec. 2009).

[9] vCloud API programming guide, Tech. Rep., VMWARE Inc., 2009.

[10] RightScale web site: http://www.rightscale.com (Dec. 2009).

[11] Scalr web site: http://www.scalr.net (Dec. 2009).

[12] WeoCeo web site: http://weoceo.weogeo.com (Dec. 2009).

[13] OGF open cloud computing interface working group, Tech. Rep., Open Grid Forum, 2009.

[14] Distributed management task force, Open Cloud Standards Incubator web site: http://www.dmtf.org/about/cloud-incubator (Dec. 2009).

[15] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, D. Zagorodnov, The Eucalyptus opensource cloud-computing system, in: Proceedings of the Cloud Computing and Applications Workshop, CCA 2008, Chicago, USA, 2008.

[16] B. Sotomayor, R.S. Montero, I.M. Llorente, I. Foster, Virtual infrastructure management in private and hybrid clouds, IEEE Internet Computing 13 (5) (2009) 14–22.

[17] B. Sotomayor, R.S. Montero, I.M. Llorente, I. Foster, Capacity leasing in cloud systems using the opennebula engine, in: Proceedings of the Cloud Computing and Applications Workshop, CCA 2008, Chicago, USA, 2008.

[18] R.S. Montero, I. Llorente, Elastic management of cluster-based services in the cloud, in: Proceedings of the First Workshop on Automated Control for Datacenters and Clouds, ACDC 2009, Barcelona, Spain, 2009, Doc ISBN: 978-1-60558-585-7, pp. 19–24.

[19] E. Huedo, R.S. Montero, I.M. Llorente, A recursive architecture for hierarchical grid resource management, Future Generation Computer Systems 25 (4) (2009) 401–405.

[20] R. Ranjan, A. Harwood, R. Buyya, A model for cooperative federation of distributed clusters, in: Proceedings of the 14th IEEE International Symposium on High Performance Distributed Computing, HPDC 2005, North Carolina, USA, 2005, pp. 295–296.

[21] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, et al., The reservoir model and architecture for open federated cloud computing, in: Internet and Enterprise-scale Data Centers, IBM Journal of Research and Development 53 (4) (2009) 4:1–4:11 (special issue).

[22] B. Miller, J. McCarthy, R. Dickau, M. Jensen, Solution deployment descriptor, Tech. Rep., OASIS, 2008.

[23] Configuration description, deployment, and lifecycle management, Tech. Rep., Open Grid Forum, 2008, URL: http://www.ggf.org/documents/GFD.127.pdf.

[24] Distributed management task force, Open Virtualization Format Specification DSP0243 v1.0.0, February 2009.

[25] F. Galán, A. Sampaio, L. Rodero-Merino, I. Loy, V. Gil, L.M. Vaquero, Service specification in cloud environments based on extensions to open standards, in: Proceedings of the Fourth International Conference on Communication System Software and Middleware, COMSWARE 2009, ACM Press, Dublin, Ireland, 2009.

[26] W. Gentzsch, Sun grid engine: towards creating a compute power grid, in: First IEEE/ACM International Symposium on Cluster Computing and the Grid, 2001, pp. 35–36.

[27] A. Andrés-Andrés, E. Gómez-Sánchez, M.L. Bote-Lorenzo, Incremental rule pruning for fuzzy artmap neural network, in: International Conference on Artificial Neural Networks, ICANN05, 2005, pp. 655–660.

Luis Rodero-Merino is a Junior Researcher with Telefónica I+D. He graduated in Computer Science at the University of Valladolid and received his Ph.D. degree at University Rey Juan Carlos, where he also worked as an Assistant Professor. Previously he had worked in the R&D area of Ericsson Spain. His research interests include computer networks, distributed systems, P2P systems and grid computing. Contact him at rodero@tid.es.

Luis M. Vaquero is a Junior Researcher at Telefónica I+D. He holds a B.Sc. in Electronics, M.Sc. in Telecom Engineering from the University of Valladolid and a Ph.D. from the University Complutense of Madrid. His research interests include distributed systems, artificial intelligence and complex systems computer simulation. Contact him at lmvg@tid.es.

Victor Gil holds an M.Sc. in Telecommunications, majoring in Telematics, from the Technical University of Valencia. After graduating, he worked as Software Platforms specialist at McKinsey & Co. He is Software Engineer at Sun Microsystems as a member of the Sun Grid Engine development team. Contact him at victor.gil@sun.com.

Fermín Galán is an R&D Engineer at Telefónica I+D. He holds a M.Sc. degree in Telecommunication Engineering from Universidad Politecnica de Madrid and is working towards his Ph.D. on Telematics in the same university. His research interests include virtualization, model-driven configuration and experimentation infrastructures management. Contact him at fermin@tid.es.

Javier Fontán is a researcher and engineer working at Universidad Complutense de Madrid. His interests are in grid and virtualization technologies. He is now part of the development team of the virtual machine manager OpenNebula that is being used in Reservoir project. Previously he has been part of EGEE and Crossgrid projects. Contact him at jfontan@fdi.ucm.es.

Rubén S. Montero is an Associate Professor in the Department of Computer Architecture at Complutense University of Madrid. His research interests lie mainly in resource provisioning models for distributed systems, in particular: Grid resource management and scheduling, distributed management of virtual machines and cloud computing. Ruben has a Ph.D. in physics (computer science program) from Complutense University. Contact him at rubensm@dacya.ucm.es.

Ignacio M. Llorente is a Full Professor and the Head of the Distributed Systems Architecture Research group at the Complutense University of Madrid. His research interests include advanced distributed computing and virtualization technologies, architecture of large-scale distributed infrastructures and resource provisioning platforms. Ignacio has a Ph.D. in Computer Science and a Executive Master in Business Administration. Contact him at llorente@dacya.ucm.es.