Service Specification in Cloud Environments Based on Extensions to Open Standards *

Fermín Galán
Telefónica I+D
Emilio Vargas 6
28043 Madrid (Spain)
fermin@tid.es

Americo Sampaio
SAP Research
CEC Belfast
americo.sampaio@sap.com

Luis Rodero-Merino
Telefónica I+D
Emilio Vargas 6
28043 Madrid (Spain)
rodero@tid.es

Irit Loy
IBM Haifa Research Laboratory
University Campus
Haifa 31905 (Israel)
loy@il.ibm.com

Victor Gil
Sun Microsystems
Dr.-Leo-Ritter-Str. 7
D-93049 Regensburg (Germany)
victor.gil@sun.com

Luis M. Vaquero
Telefónica I+D
Emilio Vargas 6
28043 Madrid (Spain)
lmvg@tid.es

ABSTRACT

Cloud computing technologies are changing the way in which services are deployed and operated nowadays, introducing advantages such as a great degree of flexibility (e.g. pay-per-use models, automatic scalability, etc.). However, existing offerings (Amazon EC2, GoGrid, etc.) are based on proprietary service definition mechanisms, thus introducing vendor lock-in to the customers who deploy their services on those clouds. On the other hand, there are open standards that address the problem of packaging and distributing virtual appliances (i.e. complete software stacks deployed in one or more virtual machines), but they have not been designed specifically for clouds. This paper proposes a service specification language for cloud computing platforms, based on the DMTF's Open Virtualization Format standard, extending it to address the specific requirements of these environments. In order to assess the feasibility of our proposal, we have implemented a prototype system able to deploy and scale service specifications using the proposed extensions. Additionally, practical results are presented based on an industrial case study that demonstrates using the software prototype how to automatically deploy and flexibly scale the Sun Grid Engine application.

Categories and Subject Descriptors

C.2.4 [**Distributed Systems**]: Distributed applications; D.3.2 [**Language Classifications**]: Specialized application languages

Keywords

Open Virtualization Format, cloud computing, service specification, interoperability

*© ACM, (2009). This is the author's version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in COMSWARE 2009, `http:\\doi.acm.org\10.1145\1621890.1621915`

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.
COMSWARE 09 June 16-19, Dublin, Ireland
Copyright 2009 ACM 978-1-60558-353-2/09/06 ...$10.00.

1. INTRODUCTION

The cloud computing paradigm has recently attracted a lot of attention from both industry and academia [23]. Cloud computing focuses on offering applications, platforms or pools of resources (e.g., processing power, CPU, networking, storage) as services in a cost effective manner. Customers of a cloud computing service in general seek the cloud offer motivated by their low costs when compared to the costs of developing and providing those services in-house. Other key factor is that computational resources are offered in a pay-per-use manner such as utilities like electricity and gas. Thus, this type of service can be very cost-effective when, for example, a huge amount of computing power for a short period is needed or when a service needs to scale up quickly as demand grows.

One quite illustrative example is the NY Times case study [18] where the newspaper IT team used Amazon EC2's service (instead of buying hardware and implementing the project in-house) to build an application to integrate their archives into their web portal. The task consisted in converting a huge number of articles saved as TIFF images and SGML files to Javascript and PNG figures and was accomplished in less then 36 hours.

There are several types of clouds [22, 24]. Amazon EC2 [1] or GoGrid [5] are Infrastructure as a Service clouds (IaaS), offering entire instances (i.e. virtual machines) to customers. Other types are Software as a Service (SaaS) and Platform as a Service (PaaS) clouds, which offer applications (e.g., Salesforce [10] and Google Apps [7]) or development and service platforms (e.g., Google App Engine [6] and Microsoft Azure [8]) respectively. This paper focuses on the interaction between the two main actors in IaaS clouds (Figure 1):

- Service Provider (SP). Companies and organizations

that are interested in “renting” capacity (e.g., processing power) from cloud infrastructure providers in order to deploy their services in a highly scalable infrastructure, due to the benefits achieved compared to using in-house infrastructure.

- Infrastructure Provider (IP). Cloud vendors that offer infrastructure (e.g., hardware) such as CPU, memory, storage, networking, etc. as a service in a pay-per use basis. One key concept of these offerings is elasticity: customers are capable of increasing/decreasing the amount of resources required depending on their needs (e.g., during peak loads more resources can be requested, while when the usage is low some resources might be released).

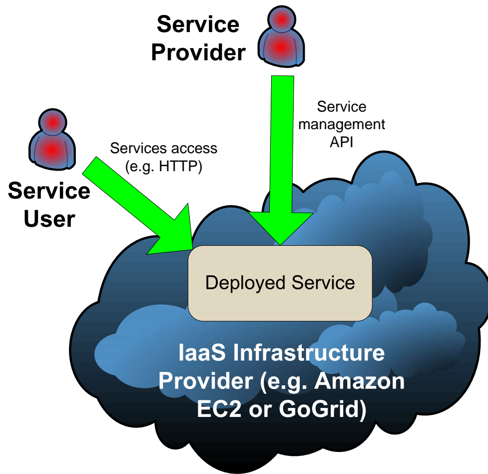

Figure 1: Cloud high-level architecture and main actors

One key element of the interaction between SPs and IPs is the service definition mechanism. In IaaS clouds this is commonly specified by packaging the software stack (operating systems, middleware and service components) in one or more virtual machines, but one problem commonly mentioned [22, 24] is that each cloud IP has its own proprietary mechanism for services definition. This complicates interoperability between clouds, which puts the SP in a vendor lock-in position. For example, if one SP has prepared his/her service to be deployed in Amazon EC2 (using the proprietary Amazon Machine Image format) then changing this service to another cloud IP (e.g., GoGrid) is not straightforward as the image creation for each provider requires several different configuration steps. Therefore there is a need for standardizing this virtual application distribution in order to avoid vendor lock-in and facilitate interoperability among IaaS clouds.

In order to solve this interoperability problem, our proposal is to base the service definition in open standards. In particular, the Open Virtualization Format (OVF) [17] is a standard recently published by the Distributed Management Task Force (DMTF)[1] that describes a specification on how to package and distribute software to be run in virtual machines. This specification initially targeted the problem of distributing software in data centers that also suffer from an interoperability problem similar to the one faced for cloud computing (data centers can have different hardware and hypervisors). Recently, it also has been considered for cloud environments [12]. OVF enables to easily package applications as a collection of virtual machines in a portable, secure and technology independent way [17]. This means that the application should be easily distributed in different data centers and interoperability problems are minimized.

[1]http://www.dmtf.org

Our contribution focuses on utilizing OVF as basis for a service definition language for deploying complex Internet applications in IaaS clouds. These applications consist of a collection of virtual machines (VM) with several configuration parameters (e.g., hostnames, IP and other application specific parameters) for software components (e.g., web server, application server, database, operating system) included in the VMs.

It is worth mentioning that we are not only proposing to use OVF as a way of solving the cloud interoperability problem. We are going a step further, addressing important issues for IaaS clouds that are not completely solved (or not considered at all) in existing cloud IPs, such as self-configuration (how virtual machines composing the service are dynamically configured e.g. IP addresses), custom automatic elasticity (how SP could specify rules and actions to automatically govern the scaling up and down of the service) and performance monitoring (how SP define the key performance indicators that are monitored by the cloud infrastructure e.g. to trigger the elasticity actions). Given that OVF has its limitations, we use extensions to the standard to achieve those goals. It is worth mentioning that our proposed extensions are backed by an actual software implementation, also described as part of this work.

In the following sections we provide an introduction to OVF as service definition mechanism (Section 2), then detail our proposed extensions for cloud environments (Section 3). In order to demonstrate the feasibility and benefits of our approach a proof of concept based on an industrial case study (Sun Grid Engine) and a service manager prototype is described in Section 4. Next, we describe related work (Section 5) and explain how our proposal goes beyond that state of art (Section 6). Finally, Section 7 closes the paper with conclusions and future working lines.

2. SERVICE DEFINITION WITH OVF

The Open Virtualization Format (OVF) objective is to specify a portable packaging mechanism to foster the adoption of Virtual Appliances (VA) (i.e. pre-configured software stacks comprising one or more virtual machines to provide self-contained services) as new software release and management model (e.g. through the development of virtual appliance lifecycle management tools) in a vendor and platform neutral way (i.e., not oriented to any virtual machine technology in particular). OVF is optimized for distribution and automation, enabling streamlined installations of VAs.

OVF specifies a format to package one or more virtual machines in a single (usually large) file[2] named *OVF package* (Figure 2 left). This file includes an *OVF descriptor* (an XML file describing the VA, in which we will focus in this paper), resources used by the VA (virtual disk, ISO images,

[2]Alternatively, package contents can be distributed as a set of files.

internationalization resources, etc.) and, finally, optional manifest and X.509 certificate files to ensure integrity and authenticity. The OVF specification describes also the *OVF environment* (a guest-host communication protocol which is used by the deployment platform to provide configuration parameters to guest at deployment time).

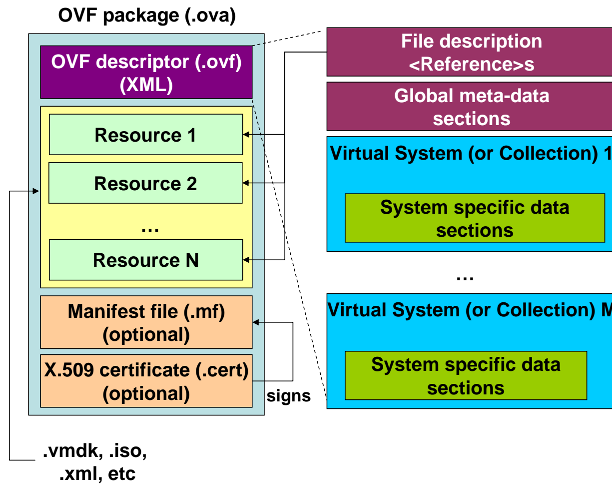

Figure 2: OVF package and descriptor structure

It is worth noting that OVF has been designed as an interoperable, neutral (regarding vendor and platform) and open standard. It is being developed within the SVPC (System Virtualization Partitioning and Clustering) working group at DMTF, which is composed by several industrial partners (each one considering their own platform and technology). New members are allowed to register and participate in their discussions as well as submit suggestions of improvements. As argued in Section 1, to base the service specification in open standards is a key requirement to ensure interoperability and reduce the SP vendor lock-in.

2.1 Brief Description

The OVF descriptor (which structure is shown at Figure 2 right) encloses information regarding the VA and its Virtual Machines (VM). It is a XML-based file (conforming to several XML Schemas included as normative documents in the specification) composed of three main parts: description of the files included in the VA (`<Reference>` tags for virtual disks, ISO images, internationalization resources, etc.), meta-data about all virtual machines contained in the package and a description of the different VM systems (single or grouped in collections).

Regarding VMs descriptions (`<VirtualSystem>`), OVF allows to specify the virtualization technology (e.g., "xen-3" or "vmx-4") and to describe the virtual hardware each VM needs (`<VirtualHardware>`) in terms of CPU, memory and hardware devices (including disks and network interfaces). The hypervisor deploying the VM must at least implement those hardware devices (note that it is allowed to implement more devices, as long as the minimum requirements are realized). In addition, ranges can be specified (e.g., for a VM a minimum of 300MB RAM and a maximum of 1204MB, being 512MB the normal amount) so the deployment platform can adjust between both ends, considering performance issues. Finally, VMs can be grouped in collections (`<VirtualSystemCollection>`) which make sense to group common information regarding several VMs.

Both meta-data and VM descriptions are structured in sections (corresponding to `<*Section>` tags). There are 10 section types (apart from the virtual hardware one described in the previous paragraph), being the most relevant ones (see [17] for details): `<DiskSection>` (describing virtual disks meta-information), `<NetworkSection>` (meta-information about all logical networks), `<ProductSection>` (specifying product information for software packages in VMs) and `<StartupSection>` (defining the virtual machine booting sequence).

Regarding OVF extensibility, new sections beyond the standard ones can be defined. There are specific semantics to state if these new sections are mandatory or not (so, in the latter case, older OVF software could process specifications including extended but not mandatory sections, simply ignoring them). This is an important property from the point of view of the work developed in this paper and we have taken advantage of it, to adapt standard OVF to cloud environment in a non disruptive way, as described in Section 3.

Apart from OVF package and descriptor, the specification 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 are specified as key-value pairs. Regarding the way to pass the environment document to the guest, OVF allows to specify the transport mechanism in a per virtual machine basis (i.e., different virtual machines can specify different transport mechanisms). Only the "iso" transport is defined up to now, consisting on dynamically generated CD/DVD images in which root directory is the XML file containing the OVF environment document. The file is read by the VM during the boot process, enabling to automatically configure application parameters (e.g., number of working threads on a web server).

3. CLOUD COMPUTING EXTENSIONS TO OVF

Standard OVF (as in [17]) was designed considering conventional IT infrastructure (i.e. data centers) as the target deployment platform, so it presents some flaws when applied directly to service deployment in clouds. Therefore, some extensions are required to address these limitations, summarized in Table 1 and described in detail in the following subsections.

Three extension mechanisms are used: adding new sections (using the mechanisms described in [17], as the ones described in Section 3.1 and 3.3.1); adding new attributes to some tags (which is not forbidden by the OVF XML Schema, as the ones in Section 3.3.2 and 3.4); and using the conventional OVF in a particular way (as in Section 3.2).

3.1 Key Performance Indicators

Services need to communicate their state to the cloud platform using a set of Key Performance Indicators (KPIs).

Table 1: OVF extensions for services specification in cloud environments

Extension	Limitation in standard OVF for cloud environments	How the limitation is solved	Extension type	Section number
Key Performance Indicators	KPI specification to trigger elasticity are not considered	`<KPIsSection>`	New section for standard OVF	3.1
Instance Index in Elastic Cluster	How to specify the instance number within a cluster at deployment time	`@VMid` macro	Special semantic for standard OVF	3.2.1
Cross-VM References	Cross-sibling references are forbidden	`@vs.prop` macro	Special semantic for standard OVF	3.2.2
IP Addresses	IP can not be assigned dynamically at deployment	`@IP(net)` macro	Special semantic for standard OVF	3.2.3
Elasticity Rules	Service elasticity is not considered	`<ElasticityRulesSection>`	New section for standard OVF	3.3.1
Elasticity Bounds	Service elasticity is not considered	`min`, `max`, `initial` attributes in `<VirtualSystem>`	New attributes in existing tag	3.3.2
Public Network Specification	Private/public nature for networks is not considered	`public` attribute in `<Network>`	New attributes in existing tag	3.4

These KPIs can be used to define elasticity rules (see Section 3.3). For example, to define a rule such as a new instance of a certain component (e.g. a web server node in a cluster) is created whenever the rate of web requests reaches a certain threshold.

Thus, each service must include a definition of all its KPIs. This is done in a new section of the OVF descriptor named `<KPIsSection>`. It just contains a list of `<KPI>` tags, each one with a single attribute `KPIname` to describe the name of a KPI. Each one of the KPIs thus defined (e.g. `petitions-LastMinute`) can be referred later in the elasticity conditions by using the prefix `@kpis` (e.g. `@kpis.petitionsLastMinute`).

When the service is active, it will send to the cloud platform in which it is running the values of its KPIs. In order to uniquely identify the service that the KPI refers to, the value must be qualified with a per-service label concatenated with the KPI name (to avoid conflict among KPIs with the same name from different services). When the cloud platform receives the KPI value, it extracts first the service it belongs to, and then updates the corresponding KPI value. On the other hand, the service components need to know the endpoint or channel to send their monitoring data to. Both (qualifier and channel) are achieved through the customization macros described in Section 3.2.4.

3.2 Macros for Deployment Time Values

The current OVF descriptor specification supports mechanisms to customize services with parameters that are known *previous* to deployment. However, in a cloud environment it is required to define certain features of the service that depend on magnitudes that are only known *at* deployment time. Note that the OVF standard already considers deployment time configuration of parameters, but only if performed manually by the user, which is not acceptable in automatic auto-scalable services we aim to build in a cloud computing environment.

An example is the customization of the service components. On many services, the configuration of some components is based on certain properties of other components which are not known before those components are deployed. For example, a service can have an application server that performs queries on a database. If the two components run on different virtual machines, the application server will need the IP address of the database host. However, its value is not known before the virtual machine where the database will run is actually deployed (for example, because it is set at deployment time through DHCP), so its IP address can not be included in the OVF descriptor unless the deployment is performed in two steps, which would break the requisite of having a totally automatic deployment of the whole service.

To address this issue, we propose to extend 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 regarding the service state at VM deployment, such as the number of instances of a certain component (e.g. nodes of a cluster of web servers) or the IP assigned to a certain host. 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 using standard transports mechanism described in Section 2.1.

Macros are used to 'weave' the relations among service components, so the service can be managed as a single entity. The following subsections describe the macros defined up to now, based on the requirements of our reference use case (Section 4). Note that the mechanism is easily extensible, so in the future more macros using the @ special character could be added, driven by new use cases. For example, `@MAC` could be used to refer to the MAC address associated to a given NIC, to be used in the configuration of MAC-based firewall rules in a given service VM.

3.2.1 Instance Index in Elastic Cluster

Consists in a simple instance counter (`@VMid`) for the number of instances of a certain component. It can be used for example to define a *ReplicaName* property with unique values for each VM in an elastic cluster of identical machines (e.g. to be configured as hostname):

```
<Property ovf:key="ReplicaName" ovf:value="Host_@VMid"/>
```

The cloud processing engine recognizes the macro `@VMid`, and will translate it to 1 for the first VM (so *ReplicaName* is "Host_1"), 2 for the second VM (*ReplicaName* is "Host_2"), etc.

3.2.2 Cross Virtual Machine References

The macro `@vs.prop` is used at `<VirtualSystemCollection>` level to refer to a property identified as *prop* (by `ovf:key`) in a different `<VirtualSystem>` sibling (i.e. different VM belonging to the same `<VirtualSystemCollection>`) identified as *vs* (by `ovf:id`). This is needed to allow cross-sibling references (which are not possible with standard OVF) but needed in some cases, as illustrated in the following example:

```
<!-- <VirtualSystem ovf:id="VMSib"> properties -->
<Property ovf:key="poolSize"
   ovf:value="50"/>
[...]
<!-- other <VirtualSystem> properties in
     the same collection -->
<Property ovf:key="localPoolSize"
   ovf:value="@VMSib.poolSize"/>
```

where the `ovf:value` of *localPoolSize* is translated to the value of the property named *poolSize* in `<VirtualSystem>` identified by *VMSib* (that is 50).

3.2.3 IP Addresses

Given that in many situations VM IPs assignment can not been known prior to deployment (because the cloud platform usually does this assignment at deployment time based on the utilization of its private IP ranges, unknown to the SP), the `@IP(net)` macro is introduced to be translated to the actual IP assigned to the VM in the virtual network *net* defined in the `<NetworkSection>` (if the VM is connected to only one network, the *net* parameter is not mandatory).

In order to clarify how it works, consider a 3-tier application with a load balancer, an elastic array of Web Servers and a Database Server. The application comprises 2 internal virtual networks and the Web Servers have two NICs (one attached to *red* network to connect to the load balancer and the other attached to the *green* network to connect to the Database Server). Networks are defined in the OVF descriptor with:

```
 <NetworkSection>
   <Network ovf:name="red" />
   <Network ovf:name="green" />
<NetworkSection>
```

The `<ProductSection>` will contain properties defined by the SP to allow customization of the Web Servers. In the example, customization is used to get IPs in the *red* and the *green* networks. In the `<ProductSection>` below 2 properties are defined (*IpInRed* and *IpInGreen*). Each describes an IP obtained from a different virtual network.

```
 <VirtualSystem ovf:id="WebServer" [...]>
   <ProductSection>
     <Property ovf:key="IpInRed"
        ovf:value="@IP(red)"/>
     <Property ovf:key="IpInGreen"
        ovf:value="@IP(green)"/>
   </ProductSection>
   [...]
</VirtualSystem>
```

According to the above service definition, for each new Web Server instance, the deployer will obtain two IP values, one from the *red* network and another from the *green* network. Let's consider that the *red* network is associated with subnet 10.0.0.0/24 and *green* with 10.0.1.0/24 in the cloud infrastructure, then the first Web Server instance will use *IpInRed* equal to 10.0.0.1/24 and *IpInGreen* equal to 10.0.1.1/24, the second *IpInRed* equal to 10.0.0.2/24, and *IpInGreen* equal to 10.0.1.2/24 and so on.

Note that actually the `@IP(net)` macro is flexible enough to cope with cases in which either the VM is performing network configuration itself (so the customization property is used to configure the VM operating system with the appropriated addressing values) or the network configurations is automatic (e.g. DHCP) but the IP value is needed to configure some software component (e.g. middleware) either in the same VM or another (using the cross-sibling reference mechanism described in 3.2.2). This last case (involving the combination of both extensions) is illustrated below:

```
<!-- <VirtualSystem ovf:id="VMMaster"> properties -->
<Property
   ovf:key="IPforMaster" ovf:value="@IP(sge_net)"/>
[...]
<!-- other <VirtualSystem> properties in
     the same collection -->
<Property ovf:key="masterIP"
   ovf:value="@VMMaster.IPforMaster"/>
```

3.2.4 Monitoring Channel

As mentioned in Section 3.1, some customization values not known before deployment time are related with KPIs. 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 (the deployment platform ensures that 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 probably unknown to the SP, the `@KPIChannel` macro is used. When the customization data is created for a new virtual machine, the cloud platform will assign the endpoint address to the properties that contain that macro.

3.3 Elasticity Specification

One of the main features of IaaS systems is the scalability or elasticity they offer (recall that Amazon denotes its cloud service as "Elastic"). These infrastructures provide SPs the capacity to scale the infrastructure allocated for their services as needed. For example, a SP can add more virtual machines if she expects peak demands, or detects a degradation on the service performance. On the other hand, if the SP detects that some resources are underused, she can ask the cloud infrastructure to withdrawn them to reduce the associated costs. This flexibility is one the main advantages of cloud infrastructures. Thanks to this feature, SPs avoid to invest on fixed hardware that could not be required in the future, and at the same time do not risk that their services suffer from a shortage of resources.

However, the SP still has to monitor periodically the state of the service to decide whether to resize the allocated resources. To avoid this, the user should be able to define *automatic elasticity rules* associated to the service. These rules, that must be supervised by the cloud infrastructure, are defined by the SP and included in the service definition. Each rule contains a *condition* (or set of conditions) to be monitored. When those conditions are met, the rule is triggered and an *action* or set of actions are executed.

The conditions to monitor will typically be associated to the state of the service and the allocated hardware resources.

The state of the service is expressed by custom KPIs, which depend on the service deployed. For example, a KPI of a database service could be the number of transactions committed in the last minute. The service deployer must include in the service definition the KPIs to monitor, and the service will be in charge of sending the KPI values to the cloud infrastructure. The state of the hardware is provided by the infrastructure itself.

A format must be specified for the declaration of the elasticity/scalability rules to be included in the service definition. This format should also allow SPs to specify bounds on the maximum and minimum amount of resources that can be allocated during the lifecycle of the service, along with the initial resources to assign at deploy time. As part of the extensions we recommend for the OVF standard, we provide a proposal for such format.

3.3.1 Elasticity Rules Format

Rules are defined inside a new section of the OVF document denoted `<ElasticityRulesSection>`. It contains a set of one or more `<Rule>` tags, each one describing a scalability rule.

Each rule is defined by four tags. The tag `<Name>` contains the rule name; the tag `<Trigger>` is used to define the conditions that when fulfilled would 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. The set of actions that can be run depends on the cloud infrastructure that parses and deploys the service. In our cloud system (Section 4.3.1) two actions are implemented: `createVM` and `removeVM`.

The following fragment contains an example of a rule definition. The example shows how macros (with a similar syntax to the ones defined for properties customization in Section 3.2) are used to define the condition and action that form the scalability rule. The condition definition uses two macros, one to refer to the value of a certain KPI (which must have been defined in the corresponding KPIs section described in Section 3.1), and the other to the amount of instances of a certain component (virtual machine). Thus, the condition can be read as follows: *when the amount of tasks in the queue size per instance is greater than 20.* Finally, the action to execute when the condition is met is to create a new instance of a certain component.

```
<Rule>
 <RuleName>Example</RuleName>
 <Trigger
  checkingPeriod="5000ms"
  condition=
  "@kpis.queueSize/
   (@components.executor.instances.amount+1)>20" />
 <Action
  run="createVM(components.executor)"
 />
</Rule>
```

3.3.2 Bounds on Resources Allocation

As mentioned above, cloud SPs should be granted with tools to define bounds on the resources that can be allocated to a service. As part of our extensions proposal, we include three new attributes to the OVF `<VirtualSystem>` tag. This tag is used to define the different systems (virtual machines) that form the service.

The new attributes added are `min`, `max` and `initial`. The two first ones set the minimum and maximum number of instances of the corresponding system that can be created, the later is used to configure the number of instances to be instantiated when the service is deployed.

3.4 Public Network Specification

OVF allows to configure the different networks that are used to interconnect service components using the `<NetworkSection>`. However, only a subset of these networks (usually just one) should be accessible to service users (typically through Internet). For example, in a 3-tier service (front-end, back-end and storage) only access to front-ends should be enabled, but not to the internal networks that interconnect front-end with back-end and storage.

Conventional OVF does not allow to specify the public/private nature of networks. Therefore, we have added the `public` attribute to `<Network>` tag to mark a network as public, as shown in the following fragment in which *external* network is public, and *internal1* and *internal2* are not.

```
<NetworkSection>
   <Network ovf:name="external" ext:public="yes" />
   <Network ovf:name="internal1" />
   <Network ovf:name="internal2" />
<NetworkSection>
```

4. USE CASE

4.1 The Sun Grid Engine Service

The proposed use case establishes the Sun Grid Engine [14], a workload management tool, as a service to execute computationally intensive tasks over the cloud computing infrastructure provided by OpenNebula [21] (an open virtual infrastructure engine), taking advantage of its scalability, flexibility and manageability.

This service has been modeled as a VA containing two images and a Service Manifest (i.e., an OVF descriptor using the extensions proposed in this paper), so the cloud infrastructure should be able to deploy, start and automatically adapt the number of service component instances to the workload upon given application-specific elasticity rules, set by the SP. Thus, the SP does not have to take care of any management task during the service lifetime. The extensions to OVF are the key elements to achieve this goal, since the configuration of SGE is not trivial in a regular physical cluster, and it gets even trickier on a cloud infrastructure, where more advanced customization is required.

An SGE service is a cluster consisting of these two types of components (Figure 3): one Master (which controls the whole Grid Engine system scheduling; managing queues, jobs, system load and user permissions) and one or more Executors (which are responsible for running jobs submitted by the user on the master).

The initial deployment comprises two VMs, the first one, called VMMaster, includes the Master and one Executor; the second one, called VMExecutor, contains just one Executor. VMMaster and VMExecutors are connected using an internal network (named *sge_net* in the figure). Additionally, an external network (*admin_net*) is used to provide a management interface to the Master component.

Once the service is running, resources are grown and shrunk according to the defined elasticity rules. The KPI used to evaluate this elasticity rule is the SGE queue length, i.e.,

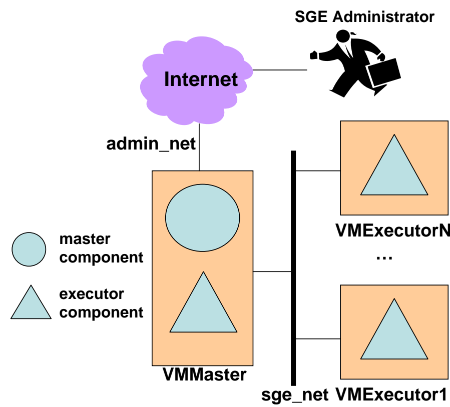

Figure 3: SGE service architecture

number of pending jobs waiting for executors to be available for them to be run on. Thus, if the queue length value divided by the current number of executors, lies over the upper threshold (e.g. 20), then a new executor VM will be deployed, started and added to the SGE cluster. Similarly, if the same parameter falls down the lower threshold (e.g. 10), then one executor is removed from the cluster.

Note that this new executor is instantiated from the same VMExecutor image file used in the initial deployment (it is a clone). Therefore it is the customization process what will make them different. The VM customization is done by a piece of software called *Activation Engine*, which runs at booting time. It parses the OVF environment document (see Section 2.1), reading it from the ISO that the deployer presents to the VM and extracting from it the parameters which then are used to perform the customization.

4.2 Service Specification

In this section we will highlight the key parts of the OVF descriptor created for the Sun Grid Engine use case. It contains the following global sections[3]:

- `<References>`. Contains references to each one of the files that are needed to deploy the service, pointing to the place the files can be obtained from (URLs, etc). In this use case there are two files, each one containing a disk image, available by HTTP.
- `<DiskSection>`. Defines the disks used by the VMs that form the service. The SGE uses only two different disk images, one for the VMMaster and another for the VMExecutors. Each `<Disk>` contains a reference to the corresponding disk image file, which was defined in the `<References>` section.
- `<NetworkSection>`. Enumerates the networks used by the service, to which VMs can be connected. The SGE considers one internal network (*sge_net*) and one public (*admin_net*), the latter using the extension described in Section 3.4. All the VM (VMMaster and VMExecutors) definitions will include interfaces to attach to the corresponding networks.

[3]For the sake of briefness only OVF fragments are included in this section. The complete descriptor can be downloaded from: `http://www.reservoir-fp7.eu/fileadmin/reservoir/presentations/comsware-sge-ovf.xml`

- `<KPIsSection>`. As explained in Section 3.1, this part of the descriptor lists the KPIs managed by the service. The SGE service only handles one KPI: the length of the jobs queue, which is managed by the master node. This KPI is labeled *queueSize*.
- `<ElasticityRulesSection>`. Described in Section 3.3, it contains the rules that define the actions to be taken depending on the service state. The scaling of the SGE service is controlled by two rules, defined as follows:
  - The first rule tries to avoid that the SGE service gets overloaded, which would impact on the QoS. By this rule, a new executor instance must be created if the amount of jobs per instance grows beyond a certain threshold.

```
<Trigger
 checkingPeriod="5000ms"
 condition=
 "@kpis.queueSize/
  (@components.executor.instances.amount+1)>20" />
<Action
 run="createVM(components.executor)"/>
```

  - The second rule tries to control the SP costs by removing instances of the executor that are not needed because of the low load. When the jobs per instance go below a certain threshold, then one instance is shut down. The instance to close is randomly chosen by the platform (this is not a problem as all executor instances are identical).

```
<Trigger
 checkingPeriod="5000ms"
 condition=
 "@kpis.queueSize/
  (@components.executor.instances.amount+1)<10" />
<Action
 run="removeVM(components.executor)"/>
```

  It is important to take into account that the amount of instances of each component is controlled by the `min` and `max` attributes of the corresponding `<VirtualSystem>` tag that defines that component. When a rule is triggered, the platform checks the amount of instances that would result of running the associated action. If that amount is lower than the `min` value, or greater than the `max` value, then the scaling action will be ignored.

Besides the tags described above, that define general characteristics of the service, the descriptor uses the `<VirtualSystemCollection>` section to contain the configuration data concerning the service components (master and executors), grouped in three different tags:

- `<StartupSection>`. It defines the deployment order of the service components. The SGE service is defined so the master is deployed before the executors.
- `<ProductSection>`. It describes the configuration data that will be passed to each VM, using the macros described in Section 3.2. Here we will define the list of `<Property>` tags that build the configuration of each component. The master configuration contains the following properties:

– Property *IPforMaster*, whose value is given by the `@IP(sge_net)` macro. Thus, it will contain the IP of the master network interface connected to the *sge_net* network.

– Property *KPIChannel*, whose value is given by the `@KPIChannel` macro, which returns the address (IP and port) of the endpoint the KPIs values must be sent to. That is, the *queueSize* KPI will be sent to that address.

– Property *KPIQualifier*, which will contain the `@KPI-Qualifier` macro value. Thus, when sending the queue length KPI value, the KPI will be identified by using the concatenation of this string with the KPI name (*queueSize*). This qualifier is created by the cloud infrastructure and is used to identify to which service the KPI refers to. Note that KPIs from several services of different SPs can be received, and so there is not guarantee that different names are assigned to these KPIs. Thus, the cloud infrastructure creates a different qualifier per each service, and the concatenation of this qualifier with the KPI name uniquely identifies each KPI.

On the other hand, the customization of the executor component is given by two properties:

– Property *MasterIP*. The value of this property is given by the macro `@VMMaster.IPforMaster`, which in turn refers to the value of the property *IPforMaster* defined in the master configuration. That is, it contains the IP address of the master[4] (this example is shown at Section 3.2.3). This is necessary, as the VMExecutor nodes (started up after the VMMaster) must contact the master immediately after booting, to register itself on the master (so notifying they are present and ready to receive jobs). Therefore the VMExecutor needs to know the IP of the VMMaster, a value that is unknown before the deployment time.

– Property *ReplicaName*. It contains the name assigned to the virtual machine, returned by the string `executor@VMId`, concatenating the "executor" token with `@VMId` macro. Thanks to it the cloud infrastructure, which keeps track of the number of VMs already started, gives the proper value (specific for this new VM) to the hostname customization parameter during the deployment.

- Finally, two `<VirtualSystem>` sections describe the hardware configuration for the VMMaster and VMExecutor components. This configuration includes CPU, memory, network interface(s) and disk(s). Each network interface is configured with the name of the network it must be attached to (which must have been defined in the `<NetworkSection>` section). For the SGE use case, an interface is attached to the *sge_net* network (VMMaster includes an additional interface to attach to *admin_net*). Also, the disk assigned to each component were defined in the `<DiskSection>` section of the descriptor. Again, both the master and the executor are configured with only one disk each.

[4]The `ovf:id` in `<VirtualSystem>` for the master is *VMMaster*.

It is worth to highlight the role of the proposed OVF extensions in this descriptor. The need of each one of the extensions for the SGE use case has been argued above (in other words, this deployment would not have been possible to carry it out without them). In addition, this use case is complete, in the sense that every extension to OVF proposed in Section 3 is used.

4.3 Service Manager Implementation

In this section we will describe the Service Manager software implementation (in Java) supporting the use case and the tasks of the main components involved, both in the Virtual Machines (Activation Engines and Probes) and in the SM itself (Parser, Lifecycle Manager and Rules Engine).

4.3.1 Deployment Architecture

In order to validate the OVF-based specification language described so far and assess the extensions developed to address the specific issues of cloud environments, we have created a Service Manager (SM) system for the deployment, monitoring and management of services. This service deployment system sits on top of OpenNebula as the infrastructure provider.

The system takes as input the service descriptions built using the extended version of OVF presented here. First, the document is interpreted by a *Parser* component. Once the document is parsed, the *Lifecycle Manager* component is notified to proceed with the actual deployment. This component controls the lifecycle of each service. First, it performs the calls to OpenNebula to deploy the components that form the service. Also, it listens to monitoring information sent by *Probes* on the service and cloud infrastructure to keep updated the state of the service. Another component (named *Rules Engine*), receives from the Parser the rules defined for the service and checks them periodically. Whenever some rule conditions are met, it triggers the actions described in the rule.

4.3.2 SGE Deployment

This Section describes the deployment process of the SGE service. It will be assumed that the two disk images (one for the master and other for the executor) are available in an external HTTP server (maybe operated by the SP). The service deployment (see Figure 4) starts when the service descriptor is passed to our system, who parses it. Then, the Lifecycle Manager component is called. This component takes care of all the deployment process. First, it outlines the components deployment order and the number of initial instances of each component (steps 1 and 2 in Figure 4). Then, it starts to deploy the service components one by one.

All components deployments follow the same steps. First, for the master component, the Lifecycle Manager prepares the customization data and makes it available by storing it into a HTTP server unaccessible from the outside (step 3). Also, it extracts and registers the service elasticity rules in the Rules Engine component (step 3). Immediately after, it calls the cloud Infrastructure Provider (OpenNebula), requesting the deployment of the virtual machine (step 4). The IP downloads the master disk image from the external HTTP server, and the customization ISO image from the platform internal HTTP server (step 5). Then, it creates

and boot the VM with the configuration requested (step 6). The master disk image, that contains the VM OS and service software, contains also the Activation Engine, which will take the configuration data from the ISO image (containing the OVF environment document described in Section 2.1) that was mounted by the cloud platform as a CD unit of the VM (step 7). Once the Infrastructure Provider has processed the deployment request and the VM boot process has started, it replies with the VM state, which includes the IP address assigned to the VM network interface (step 7). This is necessary for the Lifecycle Manager, as that data is part of the configuration of the executor VM which is deployed next.

4.3.3 SGE Monitoring and Scalability

Once the service is deployed, its KPIs are constantly monitored by the platform to trigger elasticity rules when required. Figure 5 (steps 1 and 2) shows how the probe in the master node sends the queue length KPI to the platform. The KPI is labelled $\{KPIQualifier\}.queueSize$, which is the name defined in the OVF descriptor preceded by the *KPIQualifier* assigned by the SM. Also, the probe sends the data through the end point defined in the *KPIChannel* configuration property. Both properties were passed to the master as part of the VM customization data.

The KPIs are checked by the Rules Engine component. In the SGE use case, this can lead to the triggering of scalability actions such as the creation or removal of executor nodes. Figure 5 shows an example of the interactions run when a new executor is created due to an elasticity rule. In Section 4.2 the two elasticity rules included in the SGE OVF descriptor are described. Assuming that the condition defined in the first rule is fulfilled, then the Rules Engine will run the action associated to that rule, which demands the creation of a new executor node.

The Rules Engine calls to the Lifecycle Manager (step 3) to ask for the creation of the new VM. Then, the Lifecycle Manager creates the customization data for the new executor and calls the cloud infrastructure to request the creation of the new VM (step 4). The IP will download the disk and customization ISO images (step 5) and will create and boot the corresponding VM (step 6). Inside the executor the Activation Engine reads the master IP address that is included in the customization data (it is the *MasterIP* property included in the service description) to notify the master node that a new executor is present so it can receive jobs from master (steps 7 and 8).

5. RELATED WORK

5.1 Commercial Solutions

There are several commercial IaaS cloud computing offerings. Amazon EC2 [1] is arguably the most popular and well known platform. Amazon EC2 offers a Xen-based proprietary format called Amazon Machine Image (AMI) for specifying the cloud applications. The user in general modifies pre-built public AMIs (e.g., an AMI with Linux as OS and some applications such as MySQL) or builds new AMIs from scratch which is a more complicated process [11]. Once the AMI is built the user utilizes EC2's API (which are several web services) to submit the image to the Amazon infrastructure so that it can be executed. Configuration of the applications that are contained in the AMI (e.g., web server, application server, database, etc.) is done in a manual fashion. The user logs in into its running AMI instance (e.g., through ssh) and configures the application properties by hand. There is no mechanism in the AMI for passing parameters between guest and host.

Regarding elasticity, Amazon EC2 offers a set of API functions for lifecycle operations such as starting and shutting down instances. However, the logic for monitoring parameters and taking decisions on when to grow/shrink the application is left for the EC2 user. There are companies such as Rightscale [9] that offer management services for automatically controlling elasticity on top of EC2. Rightscale offers a platform that is able to scale an EC2 application up and down based on hardware metrics. In a similar fashion Amazon offers a complementary service called Simple Queue Service (SQS) [2] that can be used to monitor load (e.g., number of simultaneous user connections, number of incoming requests, CPU, memory or bandwidth utilization) and take actions on how to scale EC2 instances.

There are several other IaaS providers such as GoGrid [5], Flexiscale [4] and Enomalism [3]. All these clouds offer similar services for deploying cloud applications in their infrastructure. They all offer their own proprietary ways for packaging and configuring the cloud applications in a similar fashion to EC2. In general the customer registers for the service, downloads some pre-built images (e.g., containing operating system, web server, databases, etc.) and customizes the images by setting application properties. All of them offer APIs (REST or web services) so that the user can take actions on the running cloud applications. Moreover, they all offer some added services (in general charged separately) such as elastic auto-scaling, management dashboards, monitoring and so on. The fact that all these different offerings have their own proprietary formats and also implement a set of common features such as elasticity in different ways favors vendor lock-in and complicates interoperability. For example, suppose that a user wants to end his/her contract from EC2 and move the application to GoGrid. This user would need to re-package his/her application in the GoGrid image format as well as re-implement the elasticity code for using GoGrid's APIs or added services.

5.2 Related Standards

There are other standards, besides OVF, that we evaluated to be used as the basis for the description of services to be deployed on IaaS clouds. This section enumerates the two most relevant ones.

Configuration Description, Deployment, and Lifecycle Management Specification (CDDLM) [19] is a format proposed by the Global Grid Forum[5] as a standard for the deployment and configuration of grid services. However, it can be used on other distributed environments. An important property of CDDLM is that it gives total control about the file format: tags names and content. It is true that by creating ad-hoc tags the SP can specify any requisite, which makes this alternative very flexible. But it also forces the SP demanding the deployment and the actual deployer IP to find a previous agreement about how to represent and interpret all the configuration parameters such as hardware requirements, instance configuration, elasticity rules, KPIs, etc.

[5]Now part of the Open Grid Forum, http://www.ggf.org

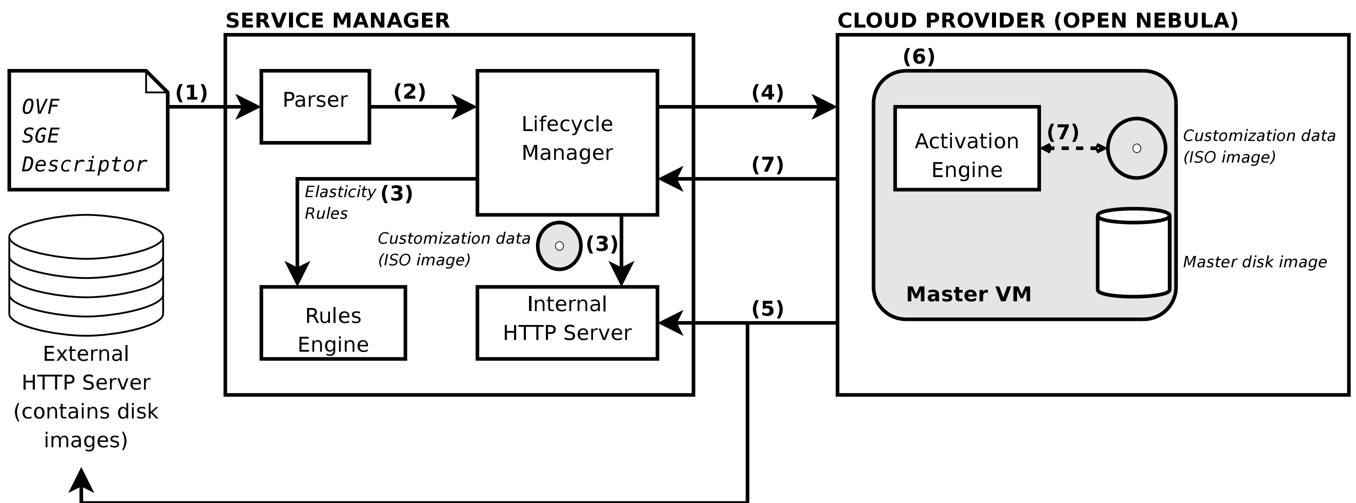

Figure 4: Deployment of Master component

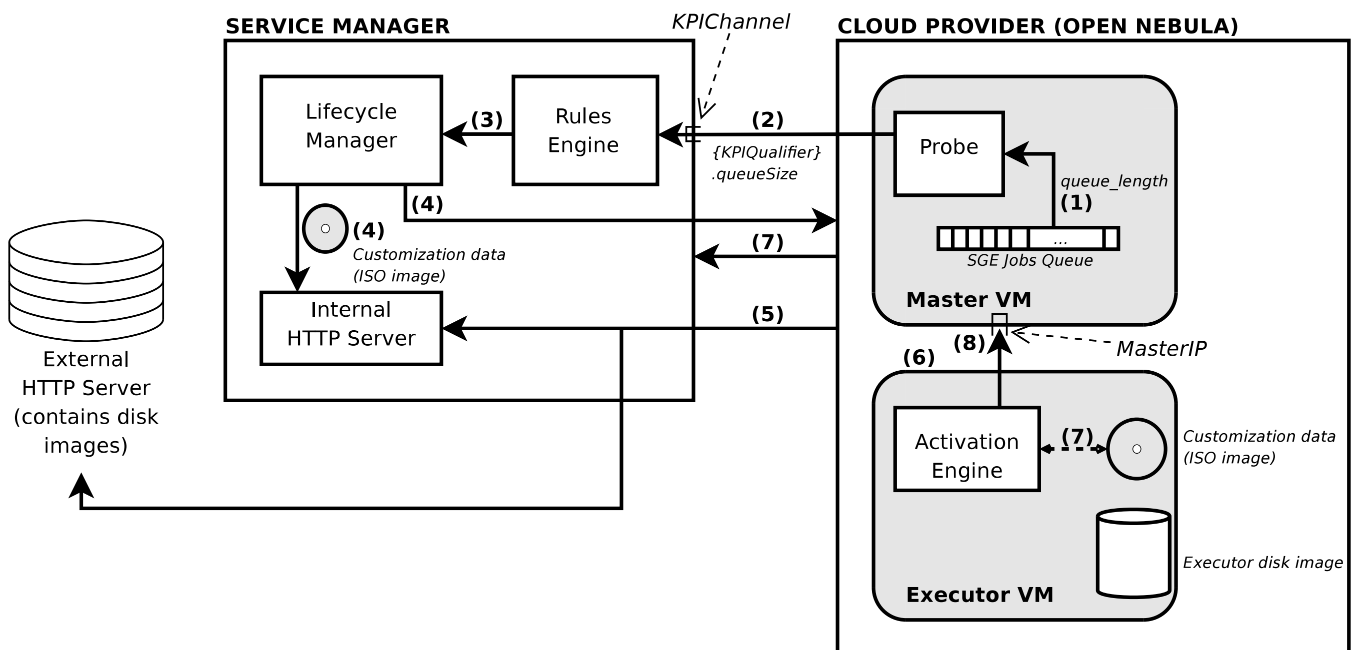

Figure 5: Monitoring and creation of new executor VM triggered by scalability rule

Another emerging standard (from OASIS[6]) is Solution Deployment Descriptor (SDD) [15]. It is oriented to the description of the management operations of software in the different stages of its typical lifecycle: installation, configuration, maintenance, etc. SDD does not address how the deployment is accomplished, nor how the configuration data must be interpreted. Thus, there must be a previous agreement about how the requisites and conditions are declared and interpreted. For example, the SDD proposal includes a *Starter Profile* [16] based on the definitions provided by the DMTF Common Information Model (CIM) [13] to promote interoperability. Finally, this format lacks 'native' features to describe the requirements closer to the infrastructure level (hardware specification).

6. DISCUSSION

In the present section, the most salient features of our OVF-based service definition language are discussed, compared with the related work in the fields of commercial cloud offerings and service specification standards performed in Section 5.

- Configuration of virtual machines. The environment properties used in OVF for passing key-value pairs enables to have self-configurable virtual machines without having the need for doing cumbersome manual configurations after launching the virtual machine such as in today's commercial clouds (Section 5.1). Moreover, our extensions allow the service specification to include values for properties that are only known during deployment time so that they can be configured during the processing of the OVF descriptor. This helps to achieve more automated and streamlined cloud application deployments when compared to current commercial clouds.

- Elasticity rules and actions. In our approach we show how elasticity rules and actions can also be included in the service definition. This information is passed to a rules engine, a component that is contained in the infrastructure (and not an added service on top of it such as in modern commercial clouds described in Section 5.1) and takes action when rules are triggered (e.g., start a new virtual machine containing the application server) when some condition holds (e.g., poor response time or high number of user requests). Elasticity is a common feature of all cloud offerings and each one provides its own way for offering this feature (normally as added services). Having a common way for specifying elasticity would help to facilitate interoperability as the elasticity logic would be transferred to the service specification and would not require changes in the cloud application logic.

- Key Performance Indicators. These properties are also defined in the service definition and indicated to the infrastructure that information about them should be collected periodically. For example the infrastructure can constantly monitor load (e.g., CPU, memory, network usage) that can be used as a basis for taking decisions such as auto-scaling of the applications. It is worth noting that our approach uses custom KPIs, that can be defined by SP at service level, which is an important advantage compared to existing mechanisms based on fixed list of hardware metrics.

[6] http://www.oasis-open.org

Regarding the other standards related with service specification, the two ones more closely related and previously analyzed (CDDLM and SDD) have limitations when considered to be used in IaaS clouds. CDDLM is not a deployment format itself, but a frame to define configuration formats. Given that the main goal of this work was to extend an already present standard, not creating a completely new format, it can be concluded that CDDLM does not suit our requirements. Regarding SDD, it is inherently flexible, but it forces SPs and IPs to reach an agreement on the deployment and data models before any deployment can be performed, while we intend the service description document to be self-contained, without the need of using third-party specifications. In addition, SDD has some expressiveness problems (in particular, to express infrastructure requirement as the `<VirtualHardware>` section in OVF does).

In summary, although some interesting combinations could be studied (e.g. to combine SDD and OVF so software description support of the former complement the latter), a deep analysis of such combinations is out of the scope of this paper and the conclusion is that between CDDLM, SDD and OVF, the latter is the most appropriated choice to base a service specification for IaaS cloud computing environments.

7. CONCLUSIONS AND FUTURE WORK

Our main contribution in this paper is a service definition mechanism based on OVF, that is not only based on open standards (thus helping to reduce the vendor lock-in in current cloud offers) but also introduces a set of extensions that address important issues such as custom automatic elasticity, performance monitoring and self-configuration. The main goal of such OVF extensions is to allow Service Providers using the Cloud to give a complete definition of a service, so it can be deployed and managed on such environment in an automatic manner. Our extensions address fundamental issues of IaaS clouds that are not addressed currently neither in existing cloud offers that are based on proprietary mechanisms nor by another standards related to software deployment (as discussed in Section 6).

Regarding completeness, the proposed extensions came up as the result of our analysis of the needs of a real service (the SGE described in 4.1). They are general enough to solve the same needs (macros for deployment time values, etc.) for any other scalable service automatic deployment in IaaS clouds. Of course, we cannot completely assure that there are no other services that could have new requirements that we have not thought about yet but in order to make the extensions as complete as possible, we are now analyzing others use cases (e.g., telco added value services as well as complex industrial ERP applications).

Utilizing a standard-based approach for deploying cloud applications is a vital step in order to achieve interoperability between clouds and avoid vendor lock-in. Moving towards interoperability in service specification is not only a benefit for Service Providers. In fact, cloud Infrastructure Providers can also benefit from adopting standards like OVF as they will be compliant with the mechanism that will probably be highly utilized by in-house clouds as more and more vendors (such as VMware, IBM or Citrix) support

this standard. Therefore, OVF compliance can potentially position cloud providers in advantage, when compared to proprietary IaaS cloud offers, as they offer a mechanism for easily importing/exporting cloud applications from in-house solutions or other OVF-compliant cloud offers.

In fact, some major industrial vendor have recently suggested to use OVF to support the deployment of appliances in infrastructure clouds [12]. However, as far as we know, the extensions described in this paper as main contribution is the first public proposal to achieve that goal. It is also worth noting that we are not just doing a theoretical proposal, given the existing Service Manager software prototype (described in Section 4.3) able to process OVF specifications using the extensions and perform the corresponding service deployment on OpenNebula based clouds.

As future working lines, the following is being considered:

- Alternatives to macros in value tags will be investigated. For example we will investigate if aspect-oriented mechanisms can be applied to weave configuration properties dynamically and in a more modular fashion similar to the work in [20] that enables to extend business processes at run time by dynamically composing web services with aspects.
- New extension to specify deployment constraints (e.g. to specify that a particular component must be deployed in a given country) and service SLAs (based on the SLAs offered by the IP, such as uptime, bandwidth, etc.).
- Adding a semantic layer on top of the OVF service definition, to be able to specify constraints at information level. This will help to ensure service configuration consistency (e.g. ensure that the KPIs used in elasticity rules are defined in the KPI section or that two elasticity rules do not contradict each other). Note that in the current implementation consistency is not enforced and the issue is left in the scope of the SP.
- Finally, considering that some of the proposed extensions could be even interesting for the DMTF (specially in the just born Cloud Incubator WG), in order to adapt the OVF official standard to cloud computing specific requirements, we plan to collaborate with this standardization body.

8. ACKNOWLEDGMENTS

This work is partially supported by the RESERVOIR EU FP7 Project through Grant Number 215605. The authors also want to thank Eliezer Levy (SAP), Stuart Clayman (University College London) and Benny Rochwerger (IBM) for their valuable help.

9. ADDITIONAL AUTHORS

Mark Wusthoff (SAP Research CEC Belfast,
email: mark.wusthoff@sap.com).

10. REFERENCES

[1] Amazon Elastic Compute Cloud (Amazon EC2). Amazon EC2, http://aws.amazon.com/ec2.

[2] Auto-scaling Amazon EC2 with Amazon SQS. http://developer.amazonwebservices.com/connect/entry.jspa?externalID=1464.

[3] Enomaly Elastic computing platform (ECP). http://www.enomaly.com/Product-Overview.419.0.html.

[4] Flexiscale cloud computing on demand. http://flexiscale.com.

[5] GoGrid cloud hosting. http://www.gogrid.com.

[6] Google App Engine. http://code.google.com/appengine.

[7] Google Apps. http://www.google.com/apps.

[8] Microsoft Azure. http://www.microsoft.com/azure.

[9] Rightscale: web-based cloud computing management platform. http://www.rightscale.com.

[10] Salesforce.com. https://www.salesforce.com.

[11] Creating an image, June 2006. Amazon EC2, http://docs.amazonwebservices.com/AmazonEC2/gsg/2006-06-26/creating-an-image.html.

[12] Citrix Drives Adoption of Virtual Appliance Portability Standard for Enterprises and Clouds, October 2008. Citrix, http://www.citrix.com/English/ne/news/news.asp?newsID=1682897.

[13] Common Information Model Infrastructure. Specification DSP0200 v2.5.0a, Distributed Management Task Force, May 2008.

[14] Guide to SGE 6.2 Installation and Configuration. Whitepaper, Sun Microsystems, Sep. 2008.

[15] OASIS Solution Deployment Descriptor. Specification, Organization for the Advancement of Structured Information Standards, Aug. 2008.

[16] OASIS SDD Starter Profile. Specification, Organization for the Advancement of Structured Information Standards, Apr. 2008.

[17] Open Virtualization Format (OVF). Specification DSP0243 1.0.0, Distributed Management Task Force, Feb. 2009.

[18] The New York Times Archives + Amazon Web Services = TimesMachine, May 2008. NY times blogs, http://open.blogs.nytimes.com/2008/05/21/.

[19] P. Anderson and E. Smith. System Administration and CDDLM. In *Proceedings of the GGF12 CDDLM Workshop*. Global Grid Forum, 2004.

[20] A. Charfi and M. Mezini. Aspect-Oriented Web Service Composition with AO4BPEL. *Lecture Notes in Computer Science*, 3250:168–182, 2004.

[21] J. Fontán, *et al.* OpenNebula: The Open Source Virtual Machine Manager for Cluster Computing. In *Open Source Grid and Cluster Software Conference*, May 2008.

[22] I. T. Foster, *et al.* Cloud Computing and Grid Computing 360-Degree Compared. *CoRR*, abs/0901.0131, 2009.

[23] L. Siegele. Let it rise: a special report on corporate IT, The Economist, 25 Oct. 2008.

[24] L. Youseff, *et al.* Toward a Unified Ontology of Cloud Computing. *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov. 2008.