Luis M. Vaquero. Long Down Av. Stoke Gifford. BS34 8QZ. +44 (0) 1173128003. luis.vaquero@hp.com

Daniel Moran. Ronda de la Comunicacin s/n 28050 Madrid. +34 91 8387641 dmj@tid.es

Fermin Galan. Ronda de la Comunicacin s/n 28050 Madrid. +34 91 8387642 fermin@tid.es

Jose M. Alcaraz-Calero. Avinguda de la Universitat s/n. 46100 Burjassot. Valencia, Spain. jose.alcaraz@uv.es

Journal of Network and Systems Management manuscript No.
(will be inserted by the editor)

Towards Runtime Reconfiguration of Application Control Policies in the Cloud

**Luis M. Vaquero · Daniel Morán ·
Fermín Galán · Jose M. Alcaraz-Calero**

Received: 15th June 2011; Revised manuscript: 14th April 2012; Second Revision: 21st June 2012; Accepted: 22nd July 2012

Abstract The main contribution of this paper is the description of an architecture for dynamically controlling the behavior of the applications deployed in the Cloud by using a set of high-level rules. This architecture is flexible enough to enable the re-definition of behavior policies at runtime. This makes it possible to adaptat the behavior of applications after deployment. It is also able to manage different cloud providers. This architecture has been implemented and the most relevant details of such implementation are also covered in this paper. Moreover, some use cases are also explained in order to provide a better description of the advantages of the proposed architecture.

Keywords: dynamic reconfiguration, cloud, rule based, service behavior, application governance.

Luis M. Vaquero
Cloud and Security Lab
Hewlett-Packard Laboratories. +44(0) 1173128003 UK, BS34 8QZ.
E-mail: luis.vaquero@hp.com
· Daniel Morán
Telefónica Digital. Telefónica. +34 91 9387641. Spain, EU 28050
E-mail: dmj@tid.es
· Fermín Galán
Telefónica Digital. Telefónica. +34 91 8387642. Spain, EU 28050
E-mail: fermin@tid.es
· Jose M. Alcaraz-Calero
Department of Computer Science. Universidad de Valencia. Spain, 46100.
E-mail: jose-maria.alcaraz-calero@hp.com
Neither the entire paper nor any part of its content has been published or has been accepted for publication elsewhere.
It has not been submitted to any other journal. JM Alcaraz-Calero was a postdoc at HP Labs when participating in this work.

1 Introduction

Information technology (IT) infrastructures may be powerful and flexible enough to cope with the maximum peak of workload expected. Traditionally, this requirement is fulfilled by means of important investments in IT infrastructure, which are idle most of the time. Nowadays, this fact is being shifted by Cloud Computing. It offers on-demand provisioning of third-party computational resources in a pay-per-use model [1]. This new way of provisioning computing resources significantly reduces IT investment in hardware infrastructures and maintenance, while optimizing resource utilization.

Cloud computing has been classified in different structural layers: i) Infrastructure-as-a-Service (IaaS) layer in charge of provisioning on-demand virtual infrastructures as an on-line service. ii) Platform-as-a-Service (PaaS) layer in charge of providing middleware services to third-parties. Such middleware services could be elasticity control services, security services, distributed data processing services, programming support services and the like. iii) Finally, Software-as-a-Service (SaaS) layer provides end-user applications that may employ the aforementioned underlying layers.

PaaS and SaaS providers usually require provisioning services for supporting automated development and configuring the virtual machines (VMs) containing PaaS and SaaS services; examples of such *provisioning services* are SLIM [2] and SmartFrog [3]. Some of the most advanced provisioning services do not only enable automated deployment of services in the cloud, but also they allow for the definition of basic scalability rules. These rules may lead to the creation and configuraton of new VMs according to some hardware metrics such as CPU, memory or disks usage.

Although current *provisioning services* are a significant step towards easy provision of PaaS and SaaS services, there are still several limitations and challenges. First, current provisioning services are not able to deal efficiently with dynamic scalability rule reconfiguration. Changes may be required not only during the initial deployment stage but also at runtime. This approach is specially demanded in cloud-based applications running for long periods of time since business requirements are time-dependent. Second, current provisioning services only cover scalability rules that are focused on increasing and decreasing the number of VMs. This simple scenario does not consider more advanced rules involving other type of service level policies. For instance, performance, quality of service, economic, or usefulness policies are not currently supported. These rules may be used to define how the application (and its components) behaves according to some context information creating adaptable applications. Finally, current provisioning services do not support architectural changes at run-time stage of the tiers available in the deployed application, which may include replacement of software components.

The main contribution of this paper is the definition of a provisioning service that addresses the aforementioned challenges, providing dynamic and runtime control of the behavior of the applications deployed in the cloud using high-level rules. The architecture proposed is based on a customized rule engine which enable the execution of the rules to govern the behavior of the cloud-based application. Thus, application providers can re-define behavior policies at runtime, adapting to new business or load conditions, which is a clear differentiating point with respect to previous approaches.

The remaining of this paper is structured as follows. Section 2 exposes related works motivating the proposed architecture. Then, Sections 3 and 4 define the domain model used for defining rules, the rules themselves and the associated languages. Section 5 describes the proposed architecture showing how to support the desired dynamism. Then, Section 6 describes some implementation details for the architecture. Relevant use cases are employed to illustrate the advantages of the proposed system in realistic scenarios in Section 7. Finally, Section 8 describes the most salient conclusions and future work.

2 Related Work

Automatic provisioning and lifecycle management of distributed applications have been intensively investigated for years. For example, Puppet [4] and CHEF [5] are good examples of client-server architectures for carrying out the automatic deployment of services in a distributed environment composed of a set of computational resources. More recently, Capistrano [6] and Control Tier [7] have appeared as extensions for Puppet and CHEF, providing service orchestration capabilities. These systems enable the management of a simple lifecycle model and to orchestrate the deployed services according to such lifecycle model.

SLIM [2], Claudia [8] and Smartfrog [3] are architectures for performing the automatic provisioning of services in the cloud. These architectures take into account on-demand infrastructure creation and configuration as part of the service deployment process. They use a basic lifecycle model to control the deployed services in the cloud. This lifecycle model is composed of basic states such as: deployed, installed, running, and destroyed service. These states have a direct implication over the VMs and the IaaS layer creating, deleting and configuring VMs in order to reach the desired status of such services. Moreover, basic scalability rules for adding/removing VMs according to hardware metrics such as CPU, memory or disks usage are also provided.

Regarding scalability features in the cloud, Amazon is already offering higher level scalability services such as Amazon Cloud Watch and AutoScaling[1]. In summary, the user statically provides some autoscaling policies based on CPU usage and other basic metrics when she is creating VMs against the IaaS API. These policies are then enforced, typically by using an elastic load balancer that is able to create and destroy new VMs registering them into the load balancer. SLIM [2] from Hewlett-Packard Laboratories also provides a similar technique using the so-called flexipoints, which are hook points where

[1] Amazon Autoscaling is available at `http://aws.amazon.com/autoscaling/`

Reference	Automatic Deployment	Orchestrated Deployment	Policy-based Lifecycle management	Dynamic Scalability	Dynamic Reconfiguration of Policies	Architectural Change
Puppet	yes	no	no	no	no	no
CHEF	yes	no	no	no	no	no
Capistrano	yes	yes	no	no	no	no
Control Tier	yes	yes	no	no	no	no
SmartFrog	yes	yes	yes	no	no	no
Claudia	yes	yes	yes	yes	no	no
SLIM	yes	yes	yes	yes	no	no
RightScale	yes	yes	yes	yes	no	no
Scalar	yes	yes	yes	yes	no	no
Our approach	yes	yes	yes	yes	yes	yes

Table 1 Comparison of different frameworks for provisioning services.

elastic load balancers are placed. Similarly, RightScale manages the creation and removal of VMs according to queues or user-defined hardware processing load metrics [9]. These approaches do not enable the alteration (reconfiguration) of policies once they are set and the application has been deployed.

Recent approaches have been designed *à la* PaaS offering a holistic mechanism for controlling how a given application should scale. These efforts are focused on controlling whole applications rather than individual VMs by setting runtime rules on how to add new VM replicas [8] and storage devices [10] of/to a given VM. Another good example of PaaS service is Scalar[2], this service offers a middleware to automatically deploy software in the cloud, i.e. SaaS components, using a graphical interface or an API.

In spite of these recent advances in application scalability control in both IaaS and PaaS layers, there is a lack of flexibility for runtime policy reconfiguration, as shown in Table 1. This dynamic change in the behavior of an application does not need to be on flexible aspects (such as elasticity), but it can also affect other aspects such as architectural behavior. In order to better understand what architectural behavior means, let us refer to the scenario recently exposed by Liu et al [11] and Wee et al [12]. These authors describe the presentation (front-end) tier of a web server. It may be better to use a load balancer to split computation among many instances for CPU-intensive web applications. However, it may be better to use a CPU-powerful standalone instance to maximize the network throughput for network-intensive applications. Yet, it may be necessary to use DNS load balancing to get around a single instance bandwidth limitation for more network-intensive web applications [11]. Then, in current provisioning services, scalability policies are statically defined and the rules cannot be dynamically changed in the rule engine to determine a new service behavior at runtime. Moreover, such policies typically apply to a single tier (front-end tier) since they are usually defined at VM (or VM "type") level. Our architecture enables the definition of policies at service-level and not only at VM level. These features are covered in this paper and they are a differentiating point of the architecture exposed, which may be an added-value to the services deployed in the cloud.

[2] Scalar is available at `https://www.scalr.net/`

3 Domain Model

This section describes the domain model used to represent applications, services and their associated VMs. This domain model is also used to describe policies by means of a policy language, which is explained in next section.

The existence of a domain model for representing the information model improves the decision making process and enables its ontological representation. This ontological representation fosters interoperability between different parties and extensibility of the domain model to include new concepts, properties, relationships and semantic information in the domain. This is similar to the Cloud Description Language described by Papazoglou [13, 14]. The reader is referred there for a detailed state of the art analysis on cloud domain models.

Several information models are proposed as *de facto* for representing services and IT infrastructures such as Common Information Model (CIM) [15], Open Information Model (OIM) [16] and Open Virtualization Format (OVF) [17]. In this proposal, a model based on OVF has been used for describing the domain model due to several reasons. i) OVF is the most widely supported and extended technology for virtualization which is directly related to the IaaS layer. ii) OVF is concise but descriptive enough for enabling the description of all the concepts needed for defining application behavior over IaaS layer. This conciseness is a nice have versus other proposals (like CIM), which contain hundreds of concepts that will probably never be used in this domain. Those concepts not included in OVF, but needed anyway for IaaS clouds, can be included using OVF built-in extensibility, as shown in [18]. iii) XML (the syntax in which OVF is based) provides an appropriate trade-off between expressiveness and computability. An information model may be represented in more expressive languages such as the Resource Description Framework (RDF) [19], Web Ontology Language (OWL) [20], OWL2 [21], etc. However, the domain model expressed in OVF does not really need these languages since XML provides enough expressiveness for describing it, while keeping the computational complexity significantly simpler than these approaches related to the Semantic Web.

According to the OVF specification, OVF only covers the distribution and deployment stages of the virtual appliances lifecycle, not the runtime stage. This is exactly our aim here and we extend this lifecycle with the inclusion of policies for describing dynamic behavior at runtime. Figure 1 depicts a simplified version of the OVF-based domain model for clarification purposes. In summary, this domain model is composed of basic concepts like *VirtualMachine*, *HardwareComponent*, *Service*, *VirtualDataCenter*, etc. used for describing infrastructures, services and their relationships. In case, the reader is more interested in a full specification of such domain model, Morán et al. [22] provide a comprehensible description.

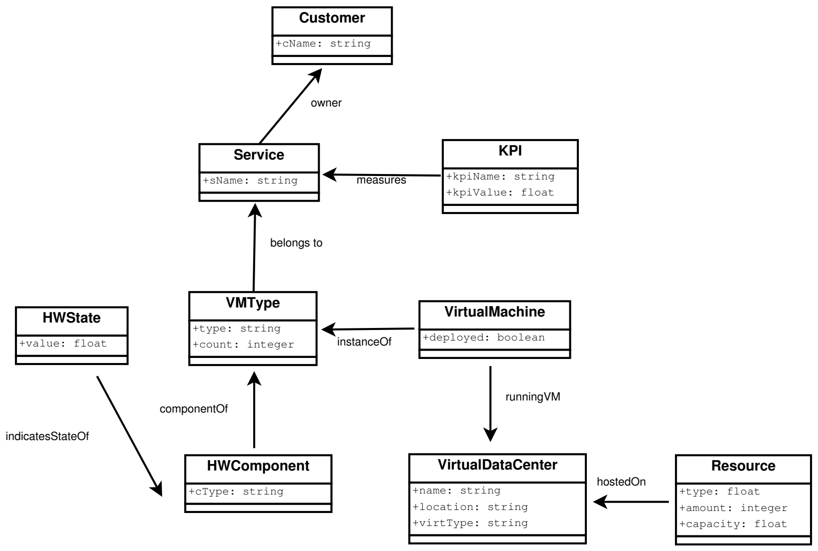

Fig. 1 Simplified OVF domain model.

4 Policy Language

While the domain model is used for describing the current information model to be managed in the cloud (composed of applications, services, VMs, storage, networks elements, etc.), the policy language is used for describing the individual and collective behavior of such elements. This is similar to the Service Constraint Language defined by Papazoglou [13,14] (readers are referred there for extensive state of the art analysis in cloud policy languages).

Given the diversity of rules to be used, the expressiveness required is really high and the chosen policy language should be capable of dealing with it. This expressiveness is generally attached to the language syntax which, in turn, is usually associated to a specific policy engine. This fact hampers portability of applications' description to different clouds. Application descriptions are not only composed of service descriptions, but also of individual and collective service behavior specifications.

Control on application behavior means facing a series of expressiveness- and interoperability-related problems. There is a wealth of policy and rule languages that could potentially be used for describing runtime application behavior across cloud environments such as Semantic Web Rule Language (SWRL) [23], RuleML[3], RIF [24], Semantics of Business Vocabulary and Business Rules (SBVR) [25] and iLog JRules [26], etc. Among all of them, Rule Interchange Format (RIF) has been specifically designed to be a portable syntax for expressing rules and policies. It is a W3C standard format that can be mapped to several rule and policy engines. These features make out of RIF

[3] RuleML specification is available at `http://ruleml.org/`

the most suitable alternative for defining application behavior in heterogeneous cloud providers and this is, therefore, the language used in this proposal (see [22] for more details). RIF assumes an underlying domain modeled using an ontological approach. This model is employed to carry out the definition of policies. The result of this choice is an expressive language able to describe semantically enriched rules by means of the underline ontological model. The only restriction for enabling interoperability between different cloud providers is that they have to share the same domain model, i.e. the OVF-based model described in Section 3.

The policy depicted in Figure 2 illustrates an example of application behavior using an extended first order logic (FOL) syntax in order to get a hint on the way rules are expressed. It is used to control elasticity over a *Tomcat* service deployed in a set of VMs. Concretely, line 1 retrieves the *Tomcat* service. Line 2 retrieves the number of VMs in which the service is deployed (*actualCount*) and the maximum range of VMs available for such purpose (*maxVMs*), this value is assumed to have been previously specified by the administrator. Line 3 retrieves the average delay associated to *Tomcat* service (*kpiValue*). Then, line 4 establishes that if a new VM can be created without violating the *maxVMs* thresholds and if the average delay taking into account a new VM is more than 300 ms, then a new VM has to be deployed, configured and executed, which is the action shown in line 6.

```
1 Service(?s) ∧ name(?s, "Tomcat") ∧ status(?s, "running")∧
2 VM(?vm) ∧ deployed(?vm, ?s) ∧ currentReplicas(?vm, ?actualCount) ∧ maxReplicas(?vm, ?maxVMs)∧
3 measuredValue(?s, ?v) ∧ name(?v, "delay") ∧ value(?v, ?kpiValue)∧
4 f : lessThan(?actualCount, ?maxVM) ∧ f : greatThan(?kpiValue/(?actualCount + 1), 300)
5 →
6 actions.createReplica(?vm)
```

Fig. 2 Example of rule-driven policy for controlling the behavior of an application.

RIF enables easy definition of high-level policies for defining application behavior on top of the static definition of the OVF-based domain model. These policies are inserted in the architecture described below in order to configure how to govern the behavior of the application deployed in the cloud.

Rule specification can somehow be seen as the definition of a finite state machine where every rule specifies a transition function that is dependent on current state and incoming events. When the specified conditions are fulfilled, the consequent action is executed (and enforced), thus triggering the transition to a new state. This latter transition could reach the same previous state, i.e. no alteration. Each state is associated with a different behavior of the application. This way of defining behavior enables the creation of custom behaviors for a given application and the selection of the appropriate behavior depending on input events.

The reader may note OVF and RIF alone are not enough to provide dynamic and automatic service behavior reconfiguration and that some clear innovations had to be made to make it work: i) appropriate mappings from high level languages such as RIF to rule-engine attached ones and, ii) adding a rich (but simple) enough semantic model that allows us to provide real meaning to the otherwise ambiguous rules. This is a clear added value of our contribution which really extends the original proposals of OVF and RIF.

5 Architecture for Managing Service Behavior in the Cloud

This section describes the proposed architecture for dynamically controlling the behavior of the applications deployed in the cloud. Figure 3 depicts an overview of the whole architecture which is mainly composed of four different components: *Rule Manager*, *Knowledge Base*, *Rule Engine* and *Rule Enforcement Module*.

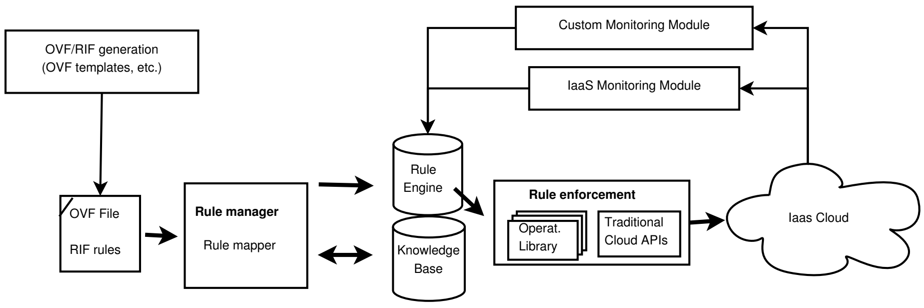

Fig. 3 Architecture overview.

A set of rules governing the applications is specified in RIF and embedded in an OVF-compliant description file. The way in which this file is generated is independent of the architecture and does not attach the architecture to any concrete scenario. For example, a PaaS application may be implemented as a set of OVF predefined templates which are instantiated in order to deploy automatically different applications on-demand in the cloud. These applications will therefore follow the behavior predefined in the RIF policies available.

A key aspect of this architecture is that the OVF-compliant file with the RIF rules embedded is inserted in the *Rule Manager* component. The component analyzes the finite state machine described by means of RIF rules and offers appropriate methods for the dynamic management of these rules. This component receives requests for changing the set of rules governing an application at runtime, usually providing a new RIF file. This new RIF is compared with the old rules performing the update of their definition, so deleting unnecessary rules and inserting the new ones. This component is a key element to enable the required dynamism in the architecture. Then, the updated RIF rules are stored in the *Knowledge Base* and mapped to the appropriate rule engine language, e.g. Drools Rule Language, Jess language, etc. for their correct execution.

As a result of the execution of the *Rule Engine*, a scheduled set of actions has to be invoked in both IaaS API and services deployed in the VMs. These invocations become the real enforcing of the application behavior. For this reason, another essential element for endowing applications with behavioral dynamism is the *Rule Enforcement Module*. This module contains a set of libraries that can be dynamically updated at runtime so that they implement the actions specified in the rules that should be executed when they are fulfilled. Deploying a new application in the cloud provider (or the same application in a different provider) may require dynamic loading of the libraries so that rule engines invoke the new operations available (application-specific library). This dynamic loading process takes advantage of programming language reflection mechanisms.

The following subsections describe in detail the different components which compose of the proposed architecture.

5.1 Rule Manager

The *Rule Manager* is in charge of managing all the rules thet define the behavior of an application at runtime. Figure 4 depicts the main modules and interfaces (only the most relevant operations are shown for the sake of understandability) are:

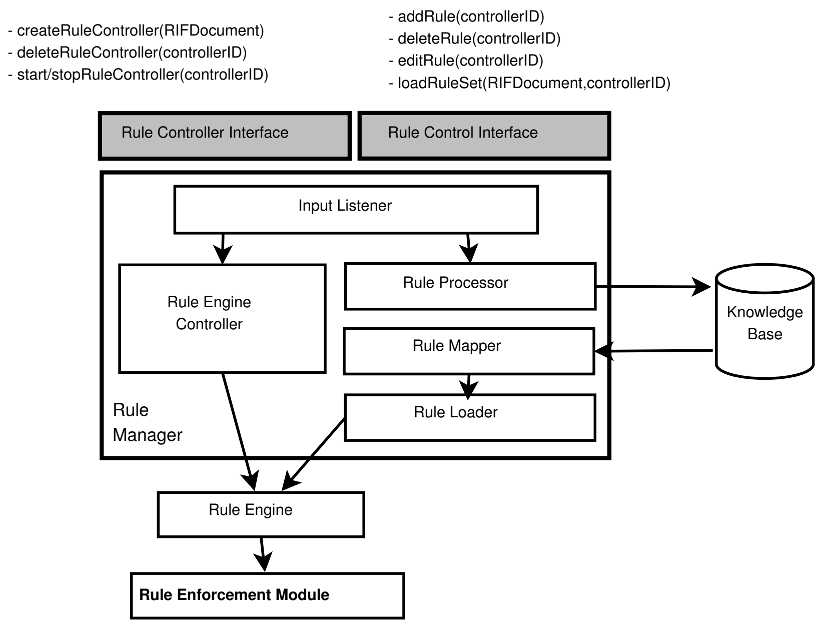
- createRuleController(RIFDocument)
- deleteRuleController(controllerID)
- start/stopRuleController(controllerID)
- addRule(controllerID)
- deleteRule(controllerID)
- editRule(controllerID)
- loadRuleSet(RIFDocument,controllerID)
Rule Controller Interface
Rule Control Interface
Input Listener
Rule Engine Controller
Rule Processor
Rule Mapper
Rule Loader
Knowledge Base
Rule Manager
Rule Engine
Rule Enforcement Module

Fig. 4 Main modules and interfaces for the Rule Manager component.

- *Rule Controller Interface.* It exposes methods related to the creation of new rule engine instances. These instances are in charge of executing the rules and inferring new information for a given application.
- *Rule Control Interface.* It exposes methods to add, edit, remove and list either individual rules or set of rules attaining a whole application. Invocations over this interface entail changes in the *Knowledge Base* in which rules are stored.
- *Input Listener.* This module is in charge of routing the incoming calls to the associated modules depending on the nature of the invoked operation.
- *Rule Engine Controller.* It receives and executes operations related to instantiation, replacement, starting and stopping of a rule engine.
- *Rule Engine.* It executes the rules provided by the *Rule Loader* module (described in detail in Section 5.2).
- *Rule Processor.* This module is in charge of splitting RIF rules received in OVF-compliant descriptors into sets of RIF rules associated to each application. This module keeps these isolated sets of rules into the *Knowledge Base*, separately. Moreover, these rules can be modified at runtime stage by mean of calls to the *Rule Control Interface.* In this case, this module stores the new rules in the correct isolation set. Since all the entities in the system are identified by unique universal identifiers (including rules, rule sets or even more fine grained entities such as the CPU of a given VM) inserting a rule into a knowledge base is pretty straightforward.
- *Rule Mapper.* It translates RIF rules into the required language by the current instance of the rule engine. In case the reader is more interested in how this map process from RIF to other rule language is carried out, Morán et al. [22] provide a comprehensible explanation of this process.
- *Rule Loader.* It is in charge of loading the rules associated to a given application into the rule engine.

5.2 Knowledge Base and Rule Engine

The *Knowledge Base* is a database repository which exposes its management and query features by means of a well-known interface. For instance, methods for executing a query against the employed domain model in the light of the facts that arrived. In multi-service scenarios (several applications controlled by the same Rule Manager) it is composed of smaller isolated knowledge bases, one per managed application. This fact enables the *Rule Processor* module to update the set of rules defined over a given application by using the appropriate knowledge base.

Regarding the *Rule Engine* module, it is composed of the current set of instances of the rule engines used to execute the set of operational rules provided by the *Rule Loader* module. The *Rule Enforcement Module* determines the appropriate set of actions to be enforced into the managed application. Usually, a rule engine only performs inference for the behavior of an application at a time, based on the RETE algorithm. However, the architecture is flexible enough to configure the number of rule engine instances available, ranging from one engine for all the applications (maximizing resource usage) to one engine per application. The set of operational rules defines a finite state machine and only those rules associated with the current state of the application are actually observed and enforced. This fact enables the selection of the current behavior to be applied over the set of possible behaviors defined in the rules.

The *Rule Engine* is usually composed of a production memory, in which rules are loaded from storage, and a working memory, which stores domain information and inferred information generated as a result of rule execution. The previously described *Rule Manager* module controls the set of rules loaded in the production memory whereas the information used in the working memory is received from:

– *OVF Descriptor.* It contains the description of domain model including infrastructure, applications and its associated services.
– *Infrastructure Measurements.* These are runtime information provided usually by a monitoring service available in the cloud provider. This information includes performance and status information about virtual and physical machines, hypervisor, application, services, etc.
– *Custom Measurements.* They are user-gathered data received from external sour-ces. This information may be used to dynamically include new measurements to build new rules on an already running application.

While information retrieval is totally heterogeneous and requires an information retrieving module per each different source of information, the update into the Rule Engine can be abstracted using a common standard interface shared between all these modules for this purpose.

5.3 Rule Enforcement

The *Rule Enforcement Module* is in charge of enforcing the actions specified by the rules when their conditions are fulfilled. Figure 5 shows an architectural overview of this module. In summary, this module exposes a programming interface called *Action* which defines the set of actions allowed for being expressed as rule consequences. This interface receives the requested actions when rules are fired sending them to the *Director*.

The *Director* is the real executor of the actions. The actions are not performed by the Rule Engine directly in order to reduce failures and avoid excessive performance degradation (especially if the action requires synchronous calls to external entities). When the *Director* receives a request, it acts as a router redirecting them to the appropriate module according to the nature of such action.

Actions can be differentiated in two types: i) *IaaS actions* which are *standard* actions over the IaaS provider API, e.g. create VM, power on VM, etc. ii)

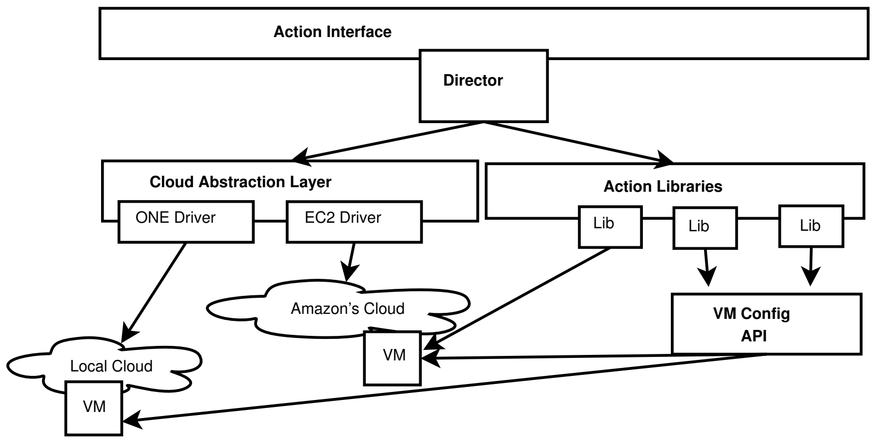

Fig. 5 Architecture of the rule enforcement module.

application-specific actions which are those operations that enable the management of a given application.

In case of IaaS actions, the *Director* invokes IaaS operations via *Cloud Abstraction Layer*. This layer is in charge of abstracting the standard IaaS operations enabling the usage of multiple cloud vendors at same time. This layer offers a basic set of IaaS operations implemented by different drivers, offering support for different cloud providers. These drivers are set up in a plug-in based architecture making the system more flexible.

With regards to the application-specific actions, the *Director* redirects the requests to a set of action libraries loaded in the *Rule Enforcement Module*. Usually, one library is loaded per application supported in the architecture providing a set of high level actions required for managing the application. All the customized actions are implemented as classes annotated with the *@ActionPlugin* annotation: this mechanism does not require this class to implement any particular interface, thus letting the service provider create its own interfaces. Then, one instance of these libraries is created on rule loading. The instance is then loaded into the Rule Engine as a set of global facts in the working memory. Classic reflection mechanisms are then employed to discover the list of actions available in the library. This way of loading the actions relies on the dynamic nature of the rule language itself, avoiding the need for compilation-time references. Therefore, the actions are automatically invoked by the Rule Engine in the action part of the rule.

Reconfiguring an application could be an example of high level actions. The *Director* again uses programming language reflection mechanisms to call the appropriate implementation of the requested action. Moreover, the set of libraries loaded in the *Rule Enforcement Module* can be modified dynamically at runtime as a plug-in structure, just inserting, removing and updating new libraries.

Almost all the application-specific actions may require connecting to VMs in order to physically execute such actions into the VM. This connection can be done at service level using client-side service libraries or can require some remote access to the VMs. In the latter case, the libraries could use the *VM configuration API*. This is an optional facility also provided in the architecture, which offers primitive actions to connect to running VMs in order to install, configure and execute services or perform any other tasks into the VMs. The API includes SCP, RDP, X11, SSH, FTP and SMB to connect to VMs and it also manages a wide range of authentication protocols such as Kerberos, user/pass, public key, X.509, or token-based authentication. This API has also been designed to be widely modular, enabling the inclusion of new non-supported methods to connect to VMs in case of necessity.

5.4 Sequence Diagram

This section describes a sequence diagram in which a user inserts a new rule in the system and after that a new incoming event is registered in the system which produces such rule to be executed. This sequence diagram shows how the different components of the architecture interact with each other in a very simple scenario.

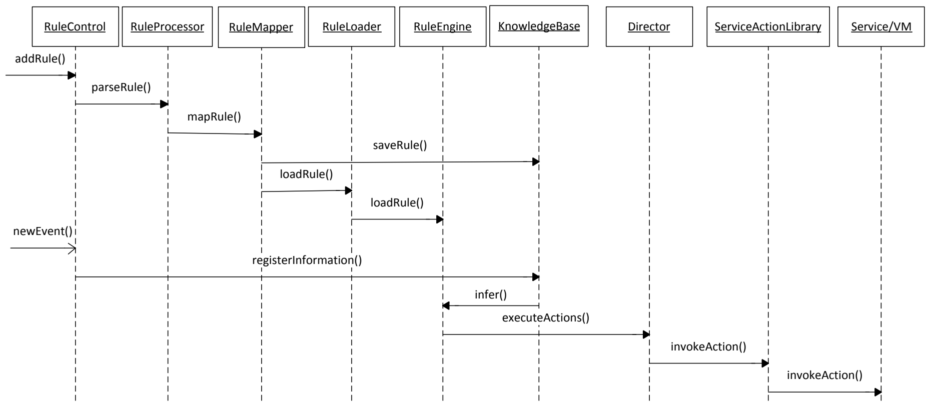

Fig. 6 Sequence diagram for adding a new rule and handling new incoming events.

Figure 6 shows the sequence diagram. First, the user inserts a new rule using the *Rule Control Interface*. Then, the rule is syntactically parsed and transformed to the syntax used by the *Rule Engine* being used. The transformed rule is loaded in the *Rule Engine* using the *Rule Loader* module. This rule is also stored in the *Knowledge Base* in order to keep everything in memory. After this, external modules in charge of detecting changes in the monitored services and infrastructures submit events to our system. When any new event is received, it is registered in the *Knowledge Base* and this triggers the

Rule Engine to perform an inference process in order to determine if any rule has to be fired. If this is the case, a set of actions is submitted to the *Director*. This director instantiates the libraries associated to such actions (or re-uses any available from a pool of pre-instantiated ones) and invokes the needed actions, enforcing the programmed behavior as specified in the rules.

6 Implementation

The architecture described in this paper has been implemented in an open-source Affero GPL-licensed prototype named Clotho[4] publically available, within the *Real Elastic Cloud* (REC) initiative at Telefónica as a natural extension of Claudia [8]. Clotho has been totally implemented using Java. This software is composed of several components, as shown in Figure 7. Firstly, an API for managing OVF and RIF files programmatically has been implemented using XML-based technologies. This API can be used by any component for authoring[5], loading and managing OVF and RIF files and for interacting with Clotho by means of a client-side REST Interface.

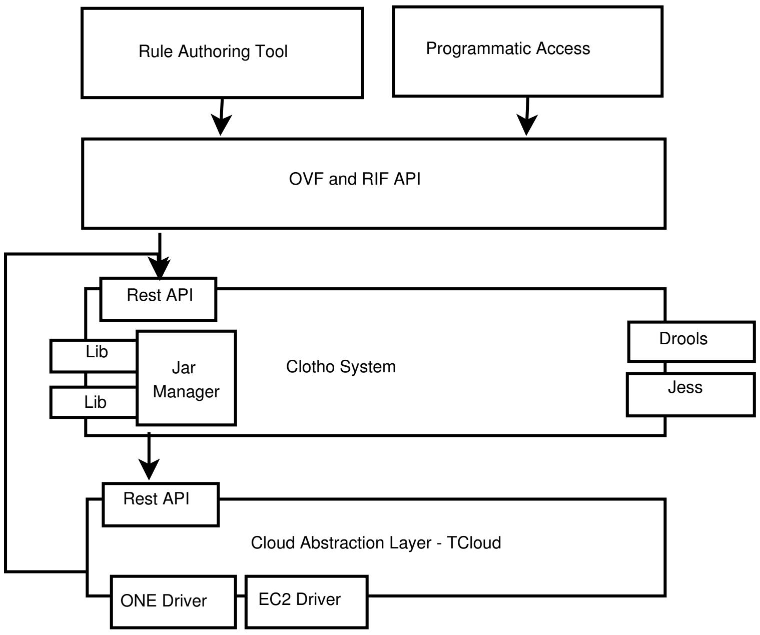

Fig. 7 Implementation overview.

Regarding Clotho, it is composed of the same modules exposed in the proposed architecture, described in Section 5. With regards to the *Rule Manager*

[4] Early prototypes are available at `http://claudia.morfeo-project.org/`

[5] The tool for creating and editing OVF and RIF is beyond the scope of this paper.

module, it offers its functionality by means of a REST interface which may be invoked by either any REST-based client or the API previously described. Currently, Clotho offers support for two different rule engines, Drools and Jess. This fact proves the efficiency of the language conversion process carried out in the *Rule Mapper* module and the flexibility of the architecture.

When action requests are received by the *Rule Enforcement* module, this module redirects requests to the proper module. In the case of IaaS actions they are redirected to the *Cloud Abstraction Layer*. This abstraction layer is implemented as a REST-interface which invokes the server-side *Cloud Abstraction Layer*, lately described. In the case of application-specific actions, they are redirected to an internal module called *Jar Manager*. This module is in charge of loading the different Java JAR libraries containing the set of actions allowed in the rule conditions. When the signed JAR files are well-located in a specific folder, it is automatically loaded at runtime by a daemon which periodically checks that folder for changes. Moreover, this module also manages removals and updates of JAR libraries. During the JAR registration process, Java reflection technology is used to extract the new set of actions publishing it in a registry. This registry is lately used by the *Director*, determining properly the final JAR on which the action is really invoked.

Clotho is built upon a REST-based cloud abstraction layer used to enable the usage of different cloud providers transparently for the users. This cloud abstraction layer has been implemented also as a public free and open source architecture coined as *TCloud*[6]. Diverse drivers have been implemented allowing *Clotho* to interact with different underlying cloud providers in a transparent manner. Concretely, an OpenNebula (ONE) [27] driver has been implemented allowing the usage of a private cloud and an Amazon EC2 driver has been also implemented allowing the usage of a public cloud.

6.1 Performance Evaluation

In order to empirically evaluate the current implementation, Clotho has been extensively benchmarked to analyze the scalability of the *Rule Manager*. In particular, testing has focused on this component since it is the most critical one in our architecture. Some of the underlying elements, such as Rule Engines are state of the art and thoroughly benchmarked in other works.

A load injector was designed to generate rule sets similar to those that would be used in complex scalability scenarios as the ones shown in this paper and described later in Section 7. The basic rule set was composed of two different states, each one implementing a different scalability policy, either vertical, upgrading or downgrading VMs based on values of Key Performance Indicators (KPI), or horizontal, creating or destroying VM instances to adapt the service to the current load. Each state contained 3 rules (i.e. 6 state-related rules in total) and 2-transition rules, for a basic set of 8 rules. The load injector

[6] TCloud specification is available at `http://claudia.morfeo-project.org/wiki/index.php/TCloud_API`

creates different rule sets replicating this basic rule set for a different layer each time, thus increasing the size of the rule set size in 8 rules each time.

Two sets of experiments were performed:

– *Fixed rules, growing KPIs* In the first one, the service was loaded with an immutable rule package and then it was fed with KPI packages of growing size, measuring processing times for each package. This experiment was repeated for several rule packages ranging from 8 to 56. The size of the KPI packages was increased exponentially from 800 to 102400 KPIs.
– *Fixed KPIs, growing rule size* In the second one, the service was loaded initially with rule packages of growing size (from 8 to 2048 rules), feeding each of them with a fixed KPI package. The experiment was repeated for several KPI packages with sizes ranging from 6400 to 25600.

All the tests have been performed in machines running Intel Core i5 2.4 GHz 4GB RAM system with Ubuntu 11.04 and JVM 1.6.0. The OS system is a clean installation and run in runlevel 3 in order to ensure that there is not other processes running in background. All the results shown in the figures are the average of 50 executions, with a mean relative error below 3%.

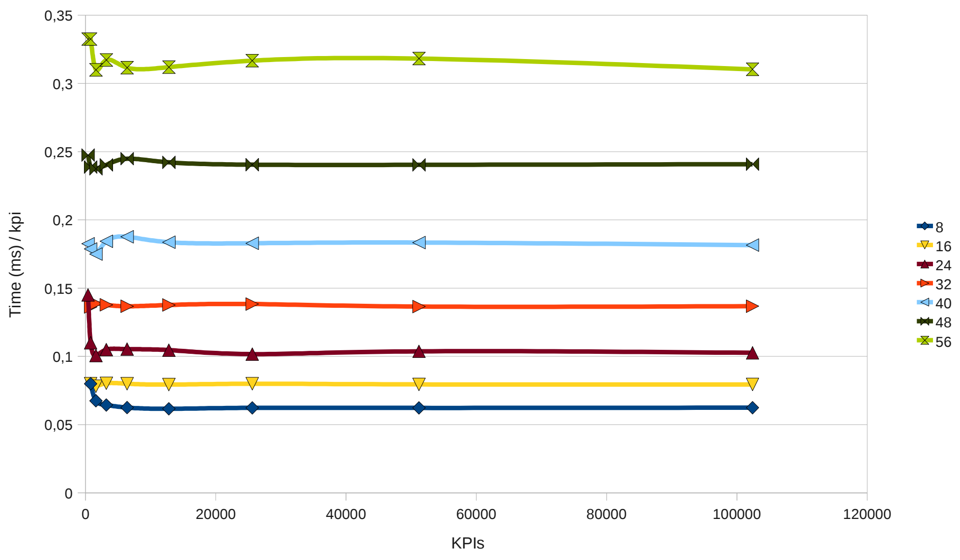

Fig. 8 KPI Processing time for packages of KPIs of growing size for differently-sized rule sets.

Figure 8 shows the results of the first experimental set, showing a slow increase in the average processing time per KPI with the increase of the number of rules. For a given rule set, the processing time remained almost constant no matter how large the input size was. This is a relevant metric in order to analyse the scalability of the proposal. Figure 9 shows the results of the second experimental set, in which the processing time grows rapidly with the number of rules. It shows an asymptotic behavior, showing small growing rates for each rule addition to large sets of rules. This ensures a good behavior for the largest sets of rules: production services typically consist of a small rule sets (in the order of tens of rules) to remain predictable/de-buggable/understandable so we expect them to be far below the biggest set of rules in the experiments (2048 rules).

These experiments show how the system performs for a single service with a complex rule package. Even in huge use case scenarios with 2048 rules and 25600 KPIs, the average processing time is under 8 min. In the production environment where the system is integrated, monitoring samples for a service are retrieved with callbacks with a frequency usually higher than that. This lets a long enough range to avoid the overlapping of rule firing processes and oscillatory behavior [8].

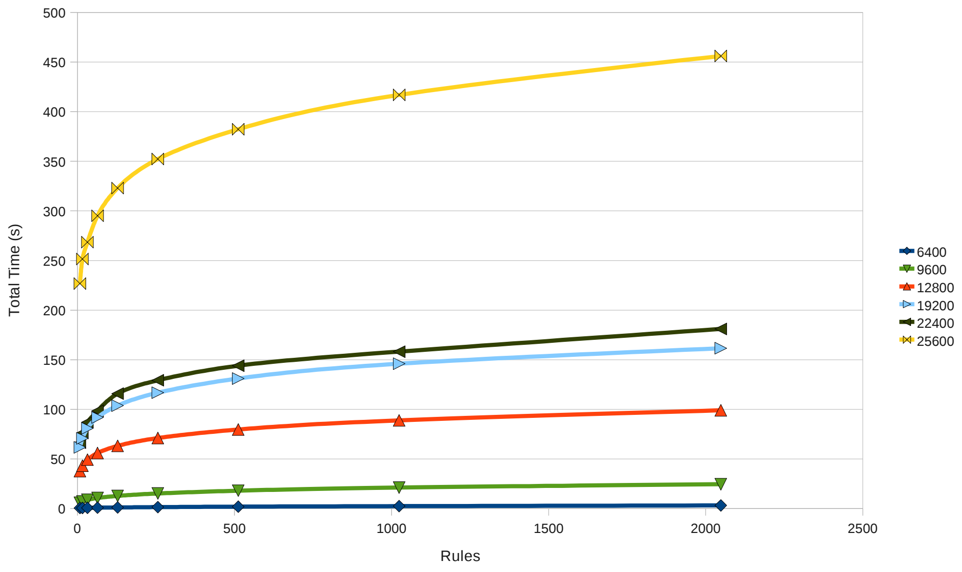

Fig. 9 Effect of the number of rules in the total processing time for fixed packages of KPIs.

Thus, the overhead introduced by the *Rule Manager* is low and it scales well with the number of KPIs and the number of rules, the underlying *Rule Engine* (shown in Fig. 4) presents some issues worth mentioning. In trying to scale the number of concurrent *Rule Managers*, we found that memory consumption of the Rule Engine was the main limitation. The Rule Engine used in our implementation (Drools Expert 5.2) in a production environment, running on Tomcat 6, triggered some issues when doing large imports of rule sets, specially derived from large compilation times for the rules. Drools has been optimized to run complex rule sets on a limited number of working memory objects

Table 2 Scalability achieved for an increasing number of services with a number of rule managers. Numbers are shown as 90th percentile of the response time in secs.

	1 Clotho	**10 Clothos**	**100 Clothos**
1 service	1.60	NA	NA
10 services	4.50	1.22	NA
100 services	40.26	3.92	1.5896
1,000 services	181.11	39.65	4.15
10,000 services	2210.03	179.02	41.29

(facts). The number of objects that makes sense varies a lot depending on the nature of these objects, but enough memory should be given to populate a RETE network in memory with space for objects, for the RETE network structure, indexes and the like. Paying careful attention to these details would prevent one from paging to disk which really improves response time. Also, the reader needs to be aware that the type and complexity of rules determine the performance of any benchmark.

For multiple services, the system creates a session per service, so giving a way for the system to divide its load between multiple processes. For this benchmark, we employed a typical 3-tiered service consisting on a set of 20 rules which combine a set of 100 different KPIs to build their condition part. In this highly frequent scenario, Clotho scales as shown in Table 2. As can be observed, a single Clotho instance is capable of handling 100 concurrent services while keeping response times in the order of tens of seconds. This response time is perfectly reasonable in cloud settings, where some responses are rather slow, e.g. booting a machine can take a few minutes, spike online services last for at least a few hours, etc. The response time of a single Clotho for 1,000 services is low (less than 4 minutes), but this may not be acceptable in some near real-time settings. Thus, virtualising Clotho to make it run on mid-sized VMs (4 vCPUs, 1GbE and 8GB RAM running Ubuntu 11.04 and Java 1.6.0) resulted in linear scalability of the response time, 100 Clothos being capable of handling 10,000 services and responding in less than 50 seconds.

7 Use Cases

This section explains some use cases implemented using Clotho in order to illustrate the advantages provided by this architecture. These use cases are described over a running scenario based on a simple three tier application composed of a Web server (Apache 2.2), an application server (Tomcat 6.0 with JONAS 5.2) and a relational database (MySQL 5.0). A load balancer can be placed in any of these tiers in order to foster elasticity of such tiers. This application is deployed in Amazon EC2 using Clotho for both deploying and managing the behavior of such application at runtime. Clotho has been configured using a set of JAR libraries for providing managing actions over

Apache, Tomcat and MySQL services. Moreover, an extra JAR has also been included in Clotho to enable the management of an HTTP Load Balancer (HAProxy) providing high level VM reconfiguration functionalities to place and remove such transparent proxy automatically.

7.1 Changing Running Environment at Runtime

The ability to change between development, testing and production environments is a typical pitfall for technically good products in order to reach a maturity phase in reasonable time. The cloud offers the chance to set up these environments on-line in the same place, reducing implementation and migration to production related inconveniences. However, different behaviors have to be configured for each of these conditions.

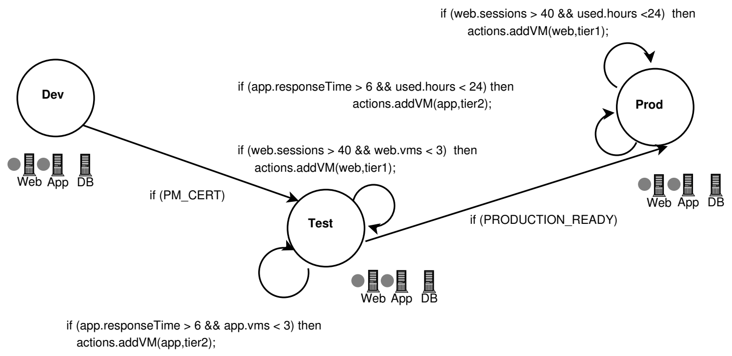
if (web.sessions > 40 && used.hours <24) then
actions.addVM(web,tier1);
Dev
if (app.responseTime > 6 && used.hours < 24) then
actions.addVM(app,tier2);
Prod
if (web.sessions > 40 && web.vms < 3) then
actions.addVM(web,tier1);
Web App DB
if (PM_CERT)
Test
if (PRODUCTION_READY)
Web App DB
Web App DB
if (app.responseTime > 6 && app.vms < 3) then
actions.addVM(app,tier2);

Fig. 10 State diagram for controlling the behavior of the application on the different running environments

Figure 10 depicts a state diagram in which these three different environments and the rules associated are defined. The small grey dots related to the machines indicate that load balancers have been placed at this tier. The aim of this use case is to show how defining different behavioral states can be useful for managing cloud applications with Clotho. In this scenario, rules are statically loaded, but they enable changing different application behavior at runtime according to the associated running environment.

Rules are defined with a *Rule Authoring Tool* and loaded in Clotho. As a result, this application is deployed into the Amazon EC2 Cloud provider. Developers perform elementary regression tests by loading new pieces of code, making sure service functionality is not damaged by new pieces, etc. No scalability is allowed in this state. The transition to test environment is performed when an administrative event reaches Clotho, i.e. the project manager certifying the product determines that it is mature enough for testing (PM_CERT). At this state, the service has limited scaling capabilities: a maximum of 2 new VMs per tier can be added or removed depending on the number of HTTP sessions on the Web server. More than 40 active sessions add a new Web server VM and configures the (transparent) load balancer to send requests to the new replica too. Similarly, a response time greater than 6 ms makes the Application tier scale.

In this real deployment scenario, Clotho responded in 0.76 sec to incoming events using a monitoring rate of 5 min (average of 50 executions with a mean relative error of 2.89%).

Similarly, when the administrative event PRODUCTION_READY is received, the application progresses to a production state in which some scaling rules control scalability in all the tiers except for the database. Now scalability is limited by *used.hours* to avoid economical denial of sustainability. This custom attribute *used.hours* represents the number of overall used hours for all the machines in the last 5 hours, e.g. 4 machines running full time in the last 5 h would account for 20 used hours. It is employed to determine the cost of a given application when running in a public cloud and it prevents scaling if the maximum available cost is exceeded.

7.2 Changing Behavior in an Already Running Application

This use case describes how Clotho allows the re-definition of rules and states governing application behavior on the fly. Figure 11 shows the initial and final behaviors after the re-definition process. Initially, a single application server is running in Amazon EC2 being managed by Clotho. Clotho monitors that the application server response time is higher than 8 ms to scale that server. However, the administrator may note that this is not an appropriate time and she decides to re-design her rules defining the final behavior.

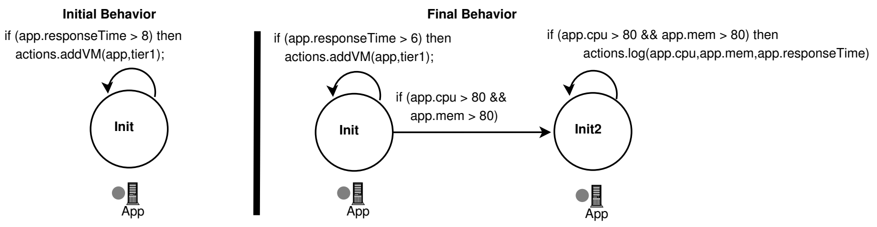
Initial Behavior
if (app.responseTime > 8) then
actions.addVM(app,tier1);
Init
App
Final Behavior
if (app.responseTime > 6) then
actions.addVM(app,tier1);
Init
App
if (app.cpu > 80 &&
app.mem > 80)
if (app.cpu > 80 && app.mem > 80) then
actions.log(app.cpu,app.mem,app.responseTime)
Init2
App

Fig. 11 Sequence diagram for changing the behavior of a running application

First, she changes the response time to 6 ms. Second, she adds a new state *Init2* for tracking CPU and memory in intensive processing periods. This point is reached when the average CPU and memory usage for all the VMs related to the application is higher than 80%. In this new *Init2* state the application provider specifies when and how to log the specific load of every application server. The rules set up are reversible. The reverse rules have not been shown for the sake of clarity.

Once the new behavior rules (i.e. final behavior) are defined by the administrator and the RIF file is received by the *Rule Manager*, these are the steps into Clotho system:

1. The *Rule Manager* (*Rule Processor* component) checks that all rule consequents can be enforced, i.e. all the necessary action JAR files are correctly loaded. If this is the case the update process continues. Otherwise, a notification is sent to the administrator and a timer is set waiting for necessary action JARs (5 min by default). If the required libraries are loaded within this time, the update process continues, otherwise, the rules are dropped. A notification is sent to the administrator and the update process is stopped.
2. The *Rule Manager* (*Rule Processor* component) overrides the knowledge base associated to such application with the new rule set.
3. The *Rule Manager* (*Rule Mapper* component) maps such RIF rules to the specific rule engine language currently running.
4. The *Rule Manager* (*Rule Loader* component) flushes the rules of the given service in the rule engine.
5. The *Rule Manager* (*Rule Loader* component) load the mapped rules to the running rule engine.

The update process is a very light-weight process (takes less than a second for all the scenarios exposed herein) to be completed. Then, there is virtually no time frame in which the service remains “unruled” (in a state of flushed old rules and unloaded new ones). The client-side API also provides fine grained rule management avoiding the need for updating the whole rule set and enabling the edition, addition and removal of single rules. On reception of unknown events Clotho just ignores them.

7.3 Application Re-tiering on the Fly

This use case describes how Clotho is able to automatically produce architectural changes on a running application. The starting scenario is a small web server running with a load balancer attached. This use case utilizes two different implementations of web servers, a light-weight web server (Jetty web server 5.1) and an enterprise web server (Tomcat 6.0). Each of these is installed in differently sized VMs that account for the different needs in terms of load and resource consumption each Web server needs. These are referred to as small and big web servers, respectively.

A set of behavior rules have been specified in order to define the scaling behavior of the initial small web server according to different conditions. These rules are shown in Figure 12. In summary, when the small server is running over more than 90% CPU usage, it scales in two different ways according to the network throughput. In case there is less than 400 Kbps of bandwidth available, it is better to scale another small web server in order to mitigate both CPU usage and bandwidth consumption together [12]. This is the case defined in *rule 1* which activates $A \rightarrow C$ transition. In this case, a new small

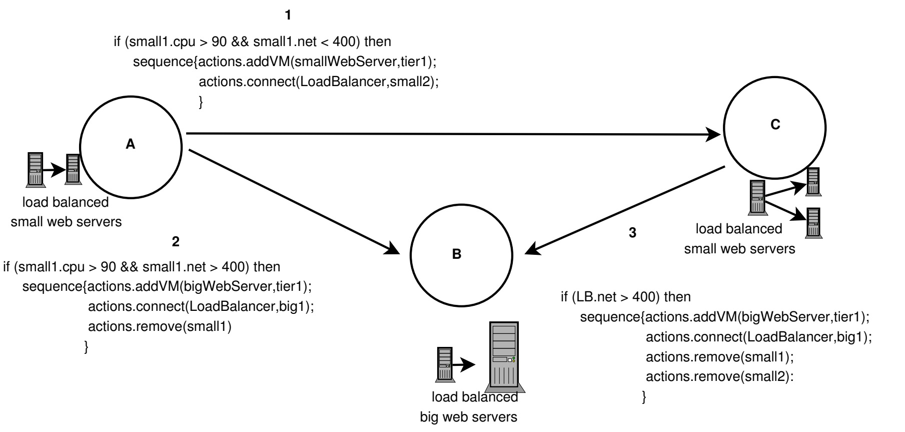
1
if (small1.cpu > 90 && small1.net < 400) then
sequence{actions.addVM(smallWebServer,tier1);
actions.connect(LoadBalancer,small2);
}
A
C
B
load balanced
small web servers
2
if (small1.cpu > 90 && small1.net > 400) then
sequence{actions.addVM(bigWebServer,tier1);
actions.connect(LoadBalancer,big1);
actions.remove(small1)
}
3
load balanced
small web servers
if (LB.net > 400) then
sequence{actions.addVM(bigWebServer,tier1);
actions.connect(LoadBalancer,big1);
actions.remove(small1);
actions.remove(small2):
}
load balanced
big web servers

Fig. 12 Changes in the architecture of a single tier over public and private clouds. Arrows and rules are in a single direction for the sake of simplicity. Analogous reverse rules were also included in this set of experiments. Reverse rules are not shown for the sake of brevity.

web server is deployed in a VM and this is attached to the load balancer in order to receive requests from the users. The other case in which bandwidth is not overloaded may require as a suitable option just a more powerful VM or a big web server application for reducing the CPU usage therein. This is the case defined in *rule 2* which activates $A \rightarrow B$ transition. In this case, the new big web server is deployed in a new VM. After that, this new server is registered into the load balancer and after that, the previous small web server is removed and its associated VM is dropped. There is not service alteration since the load balancer will transparently switch user requests to the new server. Finally, when the application is in state C, i.e. 2 small servers running and the network overload is significantly reduced, then again a big web server may be enough and even a more suitable solution for attending user requests. This behavior is configured in *rule 3* which activates $C \rightarrow B$ transition. In this case, a new big server is deployed and executed in a new VM and after that, this new server is connect to the load balancer and the previous two small web servers are dropped (and its VMs) reacting state B.

Clotho does not only support a simple homogeneous scaling of services, but it also enables architectural changes in their tiers on-the-fly. Unfortunately, state preservation is totally application dependent and our system only partially supports some state preservations (achieved already by third-party services) when applications are retiered. However, this relevant issue is being addressed by the database cloud community which is already working on this problem and our work may mature as their current research becomes accessible as a framework or a service.

In this real deployment scenario, Clotho responded in 0.6 s and the system takes around 86.1 s for enforcing completely the re-tiering of the complete web server (average of 50 executions with a mean relative error of 2.01%).

8 Conclusion and Future Work

An architecture for managing the behavior of applications in the cloud has been provided and validated successfully in this work. This architecture has been implemented in the Clotho prototype and partially released to the scientific community as a free and open source tool. Moreover, different performance tests have been executed in order to validate the scalability of the architecture and its associated implementation.

This architecture enables the usage of different rule engines and the management of different set of applications, both designed as a plug-in structure fostering the extensibility. Further, it provides a design flexible enough to use different cloud provider by means of the usage of a Cloud Abstraction Layer. This layer has been also implemented and successfully validated under different public and private cloud providers.

Regarding policy language, RIF has been identified as a good alternative for defining portable rules since it enables the translation to many other specific rule languages. Moreover, RIF has been validated as powerful enough in terms of expressivity for defining the rules provided in the different use cases presented in this paper. The OVF-based domain model presented is really able to descrie essential behavioral information about virtual appliances.

Different realistic use cases have successfully demonstrated how Clotho can considerably help in the definition and control of different application behaviors exhibiting complex reconfiguration capabilities such as dynamically changing rules and re-tiering multi-tier applications.

The developement of the system faced several difficulties, specifically in transaction management and model updates. Transaction management posed a strong integration problem, as the system has to deal with potential errors on the underlying cloud infrastructure that may leave inconsistencies in the database or the fact base. Being external (non-JTA ready) systems, standard transaction solutions did not solve these problem. That point was faced with a compensation mechanism. Concerning model updates, there are two key factors: 1) the need to use model information in production rules that would modify the underlying infrastructure this same model represents and 2) the possibility of external infrastructure modifications (e.g. the system administrator shutting down a VM for manteinance). These two factors lead to the need to create components for model synchronization to keep both the database and the fact base up to date. The action loading mechanism was also redesigned many times, before the current solution (based on JNDI lookup of the actions) emerged.

Future work includes the validation of secure code introspection techniques to allow service providers to include more JAR files without restrictions. The extension of Clotho for controlling services dwelling in application containers is also an expected step. Moreover, further experimentation should deal with some of the weaknesses we found in our tests:

– *Effect of the type of rule on the processing time.* The performance of a system based on Rule Engines will heavily depend on the kind of rules loaded

in the system. For the experiment, a small subset of KPIs was used, resembling a real service, in order to produce significant results, but a more general set of rules should be used in testing. E.g.: RETE algorithm favors pattern sharing and has a better behavior matching patterns than performing expression evaluation. Means to quantify and describe rule complexity are being explored to help to provide a strict evaluation.
– *Effect of the facts set and types of actions.* On the other hand, the nature of the KPI packages may also affect the performance. The KPIs used for the experiment were designed to produce a controlled number of state changes and action executions. Future experiments should take this into account, measuring the impact of the number of state changes and rule executions on the performance of the system.

Acknowledgements

The authors would like to thank Telefónica's team for their huge effort and professionalism beyond their duty. The authors would like to emphasize that this work has been fully developed in Telefónica within the Real Elastic Cloud (REC) initiative. In addition, authors would like also to thanks the Fundacion Seneca for sponsoring Jose Maria Alcaraz Calero under the post-doctoral program. The authors would also like to acknowledge the anonymous reviewers for their candid comments that have greatly helped to improve the quality of this manuscript.

References

1. B. Hayes, "Cloud computing," *Communications of the ACM*, vol. 51, no. 7, pp. 9–11, 2008.
2. J. Kirschnick, J. M. Alcaraz-Calero, L. Wilcock, and N. Edwards, "Towards an architecture for the automated provisioning of cloud services," *IEEE Communication Maganize*, vol. 48, pp. 124–132, Dec 2010.
3. J. Kirschnick, J. M. Alcaraz-Calero, P. Goldsack, A. Farrell, J. Guijarro, S. Loughran, N. Edwards, and L. Wilcock, "Towards a p2p framework for deploying services in the cloud," *Software: Practice and Experience*, vol. 42, pp. 395–408, 2011.
4. J. Turnbull, *Pulling Strings with Puppet.* FristPress, 2007.
5. A. Jacob, "Infrastructure in the cloud era," in *Proceedings at International O'Reilly Conference Velocity*, p. 12, 2009.
6. D. Frost, "Using capistrano," *Linux Journal*, vol. 177, p. 8, 2009.
7. D. Solutions, "Control tier," tech. rep., DTO Solutions, 2010.
8. L. Rodero-Merino, L. Vaquero, V. Gil, F. Galán, J. Fontán, R. Montero, and I. Llorente, "From infrastructure delivery to service management in clouds," *Future Generation Computer Systems*, vol. 26, pp. 1226–1240, Oct 2010.
9. RighScale, "Righscale web site," tech. rep., RighScale, May 2010.
10. H. C. Lim, S. Babu, and J. S. Chase, "Automated control for elastic storage," in *ICAC*, (New York, NY, USA), pp. 19–24, ACM, 2010.
11. H. Liu and S. Wee, "Web server farm in the cloud: Performance evaluation and dynamic architecture," in *CloudCom '09: Proceedings of the 1st International Conference on Cloud Computing*, (Berlin, Heidelberg), pp. 369–380, Springer-Verlag, 2009.

12. S. Wee and H. Liu, “Client-side load balancer using cloud,” in *SAC ’10: Proceedings of the 2010 ACM Symposium on Applied Computing*, (New York, NY, USA), pp. 399–405, ACM, 2010.
13. M. Papazoglou and W. van den Heuvel, “Blueprinting the cloud,” *Internet Computing, IEEE*, vol. 15, pp. 74 –79, nov.-dec. 2011.
14. M. Papazoglou and L. Vaquero, “Knowledge-intensive Cloud Services: Transforming the Cloud Delivery Stack ,” in *Knowledge Service Engineering Handbook* (J. Kantola and W. Karwowski, eds.), pp. 449–493, CRC Press, 2012.
15. J. W. Sweitzer, P. Thompson, A. R. Westerinen, and R. C. Williams, *Common Information Model: Implementing the Object Model for Enterprise Management.* Wiley, 2000.
16. T. Vetterli, A. Vaduva, and M. Staudt, “Metadata Standards for Data Warehousing: Open Information Model vs. Common Warehouse Metadata,” *ACM SIGMOD Record*, vol. 29, no. 3, pp. 68 – 75, 2000.
17. DMTF, “Open virtualization format white paper,” DMTF Standard DSP2017, Distributed Management Task Force, July 2009.
18. F. Galán, A. Sampaio, L. Rodero-Merino, I. Loy, V. Gil, L. M. Vaquero, and M. Wusthoff, “Service specification in cloud environments based on extensions to open standards,” in *COMSWARE ’09 (4th Conference on COMmunication System softWAre and middlewaRE)*, (New York, NY, USA), pp. 1–12, ACM, 2009.
19. F. Manola and E. Miller, “RDF prime,” w3c recommendation, W3C, 2004.
20. M. Dean, D. Connoll, F. van Harmelen, J. Hendler, I. Horrocks, D. L. McGuinness, P. F. Patel-Schneider, and L. A. Stein, “Web ontology language (OWL),” tech. rep., W3C, 2004.
21. B. Motik, P. F. Patel-Schneider, and I. Horrocks, “Owl 2 web ontology language: Structural specification and functional-style syntax,” w3c working draft, W3C, 2008.
22. D. Morán, L. M. Vaquero, and F. Galán, “Ruling the cloud: Formally specifying application behavior in a federated environment,” in *CLOUD’11 (4th IEEE Int’l Conf of Cloud Computing)*, pp. 89–96, IEEE, 2011.
23. I. Horrocks, P. F. Patel-Schneider, H. Boley, S. Tabet, B. Grosof, and M. Dean, “SWRL: A semantic web rule language combining OWL and RuleML,” tech. rep., W3C, 2004.
24. M. Kifer and H. Boley, “Rule interchange format,” w3c recommendation, W3C, 2010. http://www.w3.org/2005/rules/.
25. OMG, “Semantics of business vocabulary and rules,” tech. rep., OMG, 2007.
26. IBM, “Ibm’s ilog web site,” tech. rep., IBM, May 2010.
27. R. S. Montero and 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)*, (New York, NY, USA), pp. 19–24, ACM, 2009.

Biographies

LM Vaquero, Ph.D. was a research associate in several U.S. centers and a Researcher and IPR Manager for Telefonica. He was also a part-time assistant professor at Universidad Rey Juan Carlos. Luis is senior research engineer and patent manager at Hewlett-Packard Labs.

D. Moran, holds a M.Sc. In computer science from Universidad de Oviedo. He is research engineer with Telefonica Digital, involved in the research activities of Cloud Computing Area.

Fermn Galn Mrquez holds a M.Sc in telecommunications and a Ph.D in telematics from Universidad Politcnica de Madrid. He is currently involved in the research activities of Cloud Computing Area at Telefnica Digital. He has authored more than 40 research publications, including 5 in JCR journals, and 1 international patent.

Jose M. Alcaraz Calero received his PhD in Computer Science from University of Murcia. He has published more than 50 articles in journals, books and conferences. He is Editorial Board Member in several international journals and Organizing Committee in diverse International Conferences in the area.