Automated Configuration of Legacy Applications in the Cloud

Xavier Etchevers, Thierry Coupaye
Orange Labs
Grenoble, France
firstname.lastname@orange.com

Fabienne Boyer[1], Noel de Palma[1], Gwen Salaun[2]
[1]*University Joseph Fourier*
[2]*Grenoble INP*
LIG Labs
Grenoble, France
firstname.lastname@inrialpes.fr

***Abstract*—Current solutions for managing distributed applications in the cloud, typically covered by PaaS (Platform as a Service) offers, remain domain specific and are only partially automated. In this context, the task consisting in automatically configuring distributed applications is still a difficult issue. In this paper, we present an application architectural model and a self-configuration protocol that automates the deployment of legacy distributed applications. Our protocol is decentralized and loosely coupled to avoid the need of a global synchronization between virtual machines (VMs) during the configuration stage. An evaluation reports the performances of the protocol when applied to deploy enterprise web applications on a private cloud platform.**

***Keywords*-cloud computing; deployment; self-configuration; component;**

I. Introduction

Distributed applications in the cloud are made up of several virtual machines (VMs) that execute interconnected software elements. From a user perspective (generally an administrator in our case), deploying such applications goes through the following steps: (i) the instantiation of images, selected in the IaaS repository, as VMs in the cloud, (ii) the post-configuration of the booted VMs to set up the dynamic part of the application configuration, and (iii) the application activation, which generally requires to start the VMs in a given order so that the applicative components they embed are activated at the right time.

These configuration and activation tasks are a real burden as the VMs often include many software configuration parameters. Some of them refer to local configuration aspects (e.g. pool size, authentication data) whereas others participate in the definition of the interconnections between the remote elements (e.g. IP address and port to access a server). Therefore, once it has been instantiated, each virtual machine has to apply a set of dynamic settings in order to properly configure the distributed application. On the whole, existing deployment solutions hardly take into account these different configuration parameters, which are mostly managed thanks to dedicated (i.e. application specific) and not completely automated (i.e. human intervention is needed) scripts. Moreover, such solutions enforce many requirements that delineate the spectrum of distributed applications they can deploy. For instance, Google App Engine [1] only deals with web services that respect a restrictive programming model.

To address this issue our contribution, supported by a platform named VAMP for Virtual Applications Management Platform, is a self-configuration protocol that automates the deployment of distributed applications in the cloud. This protocol ensures three key properties that are essential, in our opinion:

1) It provides a solution aiming at self-deploying arbitrary distributed applications, independently from programming languages, programming models and conventions, runtime environments, or business domain. A main design choice is to rely on a component model in order to provide a uniform application model and configuration interface for legacy software, instead of relying on software-specific, hand-managed configuration files. Therefore, any software configured by our protocol is wrapped in a component which interfaces its administration functions (without any modification of the application code).
2) In order to avoid any centralized configuration server (e.g. Puppet configuration server [2]), the proposed self-configuration protocol is *decentralized*. Once the VMs are instantiated, the protocol is able to configure the whole application without requiring any centralized server. Therefore, each VM embeds the needed knowledge of the application model and a *configurator agent* that manages the setting of the legacy software inside the VM (thanks to a set of MBean Objects), but also participates in the global distributed configuration between the legacy software and in the application start-up.
3) The self-configuration protocol is *loosely-coupled*. Each VM starts the self-configuration protocol, just after the boot sequence, without having to care about the state of other VMs. The configuration of the distributed application will progress each time a VM belonging to the application becomes available. This avoids the need of a global synchronization between the VMs during the configuration stage. Hence it provides more scalability and agility. In order to ensure

this property, configurators send, according to the application model, part of their configuration thanks to a Message Oriented Middleware (MOM). This MOM implements a distributed message queuing system that enables configurators to exchange messages in an asynchronous and reliable way.

The rest of this paper is organized as follows. Section II describes the architectural model used by our protocol to configure a distributed application. Section III focuses on the decentralized loosely-coupled self-configuration protocol itself. Then section IV brings in the performance evaluations obtained. Section V discusses related works. Finally, section VI concludes.

II. Application Architectural Model

A cloud application can be viewed as a set of interconnected legacy software elements running on different virtual machines. A legacy application designates here an application that comes "as it is" and therefore that does not have to adapt to any arbitrary particular programming models, conventions or hypotheses to be dealt with VAMP. Three aspects are considered as essential in a configuration protocol: local configuration settings (i.e., properties values), global configuration settings (i.e. remote interconnections) and life-cycle dependencies (i.e. start order precedences). For modeling a distributed application with in terms of components, interconnections and distribution constraints within virtual machines, a component model has been used. An Architecture Description Language (ADL) [3] enables the expression of these aspects in a machine/human readable format.

A. Component model

A component model, namely the Fractal component model [4], allows to model an application from an architectural point of view. Configuration and life-cycle aspects are expressed in terms of constraints attached to the architectural elements making up a distributed application. An important aspect is that this model does not only offer a static description of a distributed application at starting time. It also provides a dynamic reification of the architectural state of the application at runtime. This makes possible to manage the deployment as an incremental task that progresses according to the dynamic state of the distributed application.

In more details, each software element, representing a deployment unit of a distributed application, is supposed to be represented by a component. Such a component mainly exposes attributes representing configuration parameters, and interfaces reflecting potential interconnection endpoints. An interface reifies either a client (respectively a server) endpoint of an interconnection that represents the classical notion of provided (respectively required) service. A client interface is characterized by a property named *contingency*. It indicates whether this interface must be connected to a server one before the component can be started (i.e. mandatory contingency) or not (i.e. optional contingency).

Bindings represent explicit interconnections between components. A binding links a client interface to a server one. It is local if the linked components are running inside the same address space (e.g. the same VM here). Otherwise it is remote. An important point is that interfaces and bindings do not make any assumption on the communication model (e.g. synchronous, asynchronous, ...) and protocol (e.g. rmi, http, ...) used by the legacy software to interoperate. Although the proposed approach relies on a component model for deployment and management purposes, it does not require applications to conform to this model. Components, interfaces and bindings only provide a way to reify and control the configuration of the legacy software elements and the establishment of their interconnections through the following configuration interfaces:

- **AttributeController**: interface for attributes management. Attributes represent the configurable properties of a legacy software element that can be observed and modified thanks to getters and setters.
- **BindingController**: interface for bindings management. Bindings represent interconnections between local or remote software elements. In order to provide a uniform view of bindings, an export/bind pattern has been used, inspired from the Reference Model of Open Distributed Processing [5][6]. This pattern has been introduced to define a way of setting up bindings by identifying and spreading configuration data used to set-up legacy communication channels. It consists of two operations called **export** and **bind**. The **export** one takes as parameters the name of a server interface (that reifies a legacy access point as described in the model). It returns an object that encodes the information required by a client to connect to the legacy access point and use it. The **bind** operation takes as parameter the name of a client interface to be bound and the exported object corresponding to the server interface. It decodes the exported object that reifies the remote access point and configures the legacy software accordingly. This is a general scheme, which is embodied in many different forms (e.g. socket, web services...).
- **LifecycleController**: interface for software elements life-cycle management. This interface provides an explicit control over a component life-cycle, through start and stop methods, hence over the legacy software element wrapped by this component.

To illustrate the use of these configuration interfaces, let's consider a simple example of a component representing an Apache server, wrapping the legacy configuration file and scripts of the Apache server as follows:

- The attribute interface is used to set attributes related to the local execution of the Apache server. For instance,

a modification of the *port* attribute of the Apache component is reflected in the *httpd.conf* file in which the port attribute is defined.

- The binding interface is used to connect the Apache server with other middleware tiers. For instance, the bind operation on the Apache component sets up a binding between an instance of Apache and an instance of a servlet container (e.g. Tomcat). The invocation of this bind method is reflected at the legacy layer in the *worker.properties* file used to configure the connections between the Apache and Tomcat servers.
- The life-cycle interface is used to start or to stop the server as well as to read its state (i.e. running or stopped). It is implemented by calling the Apache commands for starting/stopping a server.

B. Architecture Description Language

In order to offer higher level control abstractions compared to specific configuration scripts for managing a legacy application, it is necessary to get an architectural description of the application. In our approach, this description is component-baed. It enables notably the representation and the reification of the application. Its specification is based on an ADL (namely the VADL for VAMP ADL), that extends the standard language OVF (Open Virtualization Format [7]) dedicated to virtual machines description. This extension consists in an architectural view of the application distributed within the VMs of an OVF package. It conforms to the Fractal ADL[1] associated to the Fractal component model.

The VADL description of a distributed application consists of a XML-based structure (OVF and Fractal ADL are both XML-based) that encompasses the notions of components, attributes, interfaces, and bindings. While adding the notion of VM to each component description (thanks to the specific tag *virtual-node*), VADL also permits to describe the distribution constraints of components within virtual machines.

To illustrate this ADL, we made use of the Java 2 Enterprise Edition Platform (JEE), which defines a model for developing web applications in a multi-tiered architecture. Such applications usually receive requests from web clients, that flow through a web server, then to an application server to execute the business logic of the application and generate web pages on-the-fly, and finally to a database that persistently stores data.

The following VADL description models a simple infrastructure made up of an Apache web server connected to a JOnAS application server itself bound to a MySQL database. Each component describes its client and server interfaces (<interface .../>), their binding (<binding .../>), configuration attributes (<attribute .../>) and the component implementation used to control the software (<content .../>). Moreover each component references the virtual image that contains its software (<virtual-node .../>). The virtual image is defined using the standard OVF section (<VirtualSystem .../>)

```
<Envelope ...>
 ...
 <!-- Applicative architecture -->
 <AppArchitectureSection name="JEE">
  <component name="Apache1">
   <interface name="AJP13" role="client".../>
   <content class="fr.orange.ApacheMBean" .../>
   <virtual-node name="VM0"/>
  </component>
  <component name="AppServer1">
   <interface name="AJP13" role="server" .../>
   <interface name="BD" role="client" .../>
   <content class="fr.orange.JonasMBean" .../>
   <virtual-node name="VM1"/>
  </component>
  <component name="MySql1
   <interface name="BD" role="server" .../>
   <content class="fr.orange.MysqlMBean" .../>
   <virtual-node name="VM3"/>
  </component>
  <binding client="Apache1.AJP13"
       server="AppServer1.AJP13" />
  <binding client="AppServer1.BD"
       server="MySql1.BD" />
 </AppArchitectureSection>
  ...
  <!-- Virtual machines configuration -->
  <VirtualSystemCollection ovf:id="App">
    <!-- VM0 configuration -->
    <VirtualSystem ovf:id="VM0" rsrvr:min="1"
                   rsrvr:max="1">
       ...
    </VirtualSystem>
    ...
 </VirtualSystemCollection>
</Envelope>
```

III. Protocol Description

Being given a distributed application description in the previous formalism, all the virtual images composing this application are first generated and instantiated within VMs by a tool chain driven by VAMP [8]. Each VM embeds the VADL description of the distributed application, as well as a *configurator agent* that enables the self-configuration protocol detailed in this section. Configurators are simple Java objects that behave according to an event/reaction model. All configurators evolve in parallel and each of them carries out three tasks in sequence:

1) Based on the application architectural model contained in the VADL descriptor, it creates the local applicative components and configures them.
2) It participates in the global application configuration by inferring the remote bindings. A remote binding associates a client (respectively server) interface of a local component with a server (respectively client) interface provided by a component located in another virtual machine. In such a situation, both virtual machines need to interact together in order to set up this binding.
3) It starts the local applicative components. A component can be started only when each of its client

[1] http://fractal.ow2.org/fractaladl/

interfaces with mandatory contingency is bound to a server interface of a started component.

Both steps 2 and 3 introduced above require communications to be established between the different configurators. These exchanges are carried out thanks to messages sent through the MOM that provides an asynchronous communication model. Thanks to this communication model, configurators do not need to be both ready for execution at the same time and tolerate transient network failure. Consequently the configuration protocol can be designed and implemented in a time-independent manner. This property allows each VM to boot and to start the configuration process independently. The global configuration will progress each time a VM becomes available.

The steps for establishing the remote applicative bindings (step 2 above) and then for activating the components (step 3 above) are organized according to the following protocol executed sequentially by each configurator:

a. For each binding linking a client side component C_1 to a server side component C_2, the configurator K_2 (responsible for C_2) exports the C_2 server interface to configurator K_1 (responsible for C_1). Then, when K_1 receives such an exported interface, it proceeds with the C_1 local binding configuration.
b. Once the configurator has exported all its server interface, it can launch the process for starting the applicative components. It consists first in activating all applicative components that do not own any binding with a mandatory contingency. Then, the configurator determines, for each activated component, the list of the bindings whose server side is local. For each of them, it sends a *start* message to each configurator responsible for the associated client side.
c. Upon receiving a *start* message, a configurator determines whether the component targeted by the message has all its server interfaces with mandatory contingency satisfied (i.e. that the corresponding components are started). In such a case, the component is started and the configurator sends a *start* messages to the configurator having components bound to the new started component.

The overall behavior of a configurator follows a precise workflow that is summarized in Fig. 1 where actions (`CREATEVM`, `CREATECOMPO`, etc.) appear in boxes identified using natural numbers (❶, ❷, etc.). Diamonds stand for choices, and each choice comes with a list of box identifiers that can be reached from this point.

First, the configurator starts (❶). Then, it successively creates all the components described in the ADL for this virtual machine (❷), binds local components (❸), and sends binding messages to remote components (❹). Diamonds propose different choices in the workflow because a virtual machine has not necessarily local bindings for instance, and

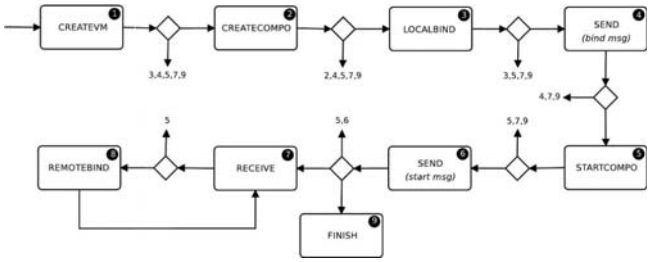

Figure 1. Abstract view of the configurator process

in such a case the configurator jumps to the next step.

Next, the configurator activates its local components that can be started (❺). At that moment in the protocol execution, only the components without any mandatory client interfaces, or those whose all mandatory client interfaces are connected to local components, can be started. From then on, for each component C_{server} started, the configurator sends to every remote component connected to it through an application binding, a start message (❻) indicating to the remote component that C_{server} is started. When the configurator has started all the local components that can be launched, it starts reading from its input communication channel provided by the underlying MOM (❼). Two kinds of message can occur: (i) upon receiving a binding request message, the configurator binds the local component to the remote one (❽), (ii) upon receiving a message indicating that a remote component has been started, the configurator keeps track of this information and goes back to ❺ in order to check whether other local components can be started (those with all mandatory client interfaces connected and whose corresponding server components is started) or not.

Fig. 2 provides an example of an application (left part of the figure) and the corresponding self-configuration protocol execution (right part of the figure). This example involves three VMs having interconnected components. The diagram associated with the self-configuration protocol's execution shows the communication exchanges between the VMs configurators. After being started, each VM creates its local components and process their local bindings if any. Then, as shown in the diagram, VM3 exports to VM2 the C4 server interface. This implicitly means the ADL describing the application indicates that a component of VM2 has a client interface that shall be bound to a server interface of C4 in VM3. In the same way, VM2 exports to VM1 the C3 server interface. After a VM has exported all the needed server interfaces to the other VMs, it sends the start messages, indicating that a given component has just been started. A main point shown by the diagram is that a VM can send information to other VMs without knowing about their state. Even if some of these VMs are not running (e.g., they have not been instantiated yet), the asynchronous communication model provided by the underlying MOM keeps message

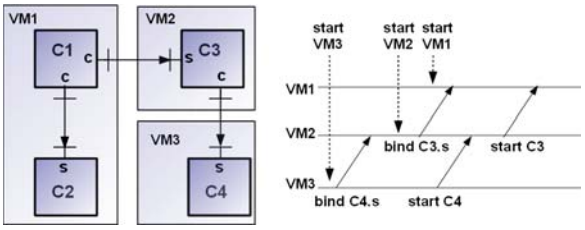

Figure 2. Application configuration (left) and self-configuration protocol (right)

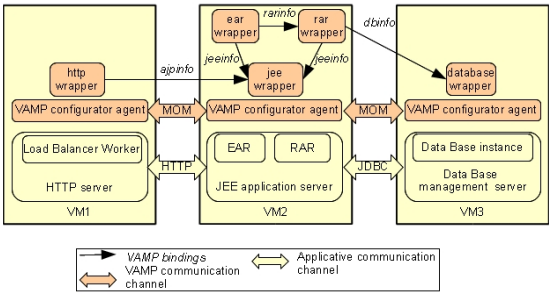

Figure 3. Architecture of the JEE enterprise web application used for benchmarking the VAMP deployment process

pending until they can be delivered properly.

IV. Evaluation

The goal of this assessment is to evaluate the VAMP system efficiency for deploying an application in the cloud. It focuses only on the performances in terms of deployment speed.

A. Use Case

The application used to bench the VAMP deployment process is an enterprise web application that aims at managing the product references, the catalogs, the offers and the markets of a company. It consists of:

- a MySQL database server with the instance of the database gathering the applicative data. Both of these entities deployment is managed by a single VAMP wrapper component (namely the *database wrapper*);
- a JEE JOnAS application server that instantiates the business logic application (ear) and a JDBC connector to the database. A VAMP wrapper component is associated to each of these entities (respectively the *jee wrapper*, the *ear wrapper* and the *rar wrapper*);
- a HTTP Apache front-end server that routes the user requests to the JEE application server. Its associated VAMP wrapper component is called the *http wrapper*;

Each of these three subsystems is installed in an own 4 GB virtual image that VAMP will instantiate within a virtual machine (see Fig. 3[2]). Each one embeds one virtual CPU. Both VMs running the database server and the HTTP server own 128 MB virtual memory whereas the virtual machine dedicated to the JEE application server has 512 MB virtual memory.

Each client interface of each applicative component has a mandatory contingency.

Table I illustrates, for each applicative binding, the configuration data exported by the server interface to the client one.

[2]On this figure, the name of both the client and the server interfaces of each binding is the same. It is so just referred once on the binding in italic

B. Tests Environment

The validation environment that has been used is a private cloud platform. It consists of Dell Studio Hybrid machines (1 x Intel Core 2 Duo T8100 2.1 GHz, 4 GB RAM and 320 GB HDD) linked through an Ethernet 100 Mbps local area network. Each machine is installed with a Linux operating system (Debian Lenny 64-bits), a Xen hypervisor (v3.2) and a proprietary IaaS platform. One of the physical machines (namely the *IaaS manager*) is dedicated to the IaaS platform administration. The remaining physical servers make up a cluster of *instantiation nodes* or *deployment nodes*.

The only noticeable behavior of the proprietary IaaS platform used, is its *placement policy*. This one ensures that each instantiation node always embeds the same number (+/-1) of VMs of each different type (i.e. data base server, application server, http server). This way, the load applied to each instantiation node in terms of resources consumption (CPU, memory, I/O, ...) is roughly equivalent.

However, insofar as the VAMP platform is part of the PaaS layer, it is independent from any IaaS platform. Thus the benchmark of the self-configuration it provides, does not focus on the IaaS behaviors efficiency. For each instantiated virtual machine, the different assessment metrics are evaluated after the VM finished to be provisioned on an instantiation node (see the following subsection for details).

Table I
Configuration data shared between applicative components

Client cmp	Client itf	Server cmp	Server itf	Shared configuration data
rar wrapper	dbinfo	database wrapper	dbinfo	IP address, database name, database user and password
rar wrapper	jeeinfo	jee wrapper	jeeinfo	IP address, JOnAS home directory, JOnAS working directory, JOnAS instance name, JOnAS domain, Java home directory
ear wrapper	jeeinfo	jee wrapper	jeeinfo	IP address, JOnAS home directory, JOnAS working directory, JOnAS instance name, JOnAS domain, Java home directory
ear wrapper	rarinfo	rar wrapper	rarinfo	IP address
http wrapper	ajpinfo	jee wrapper	ajpinfo	IP address, AJP port, JVM route

C. Results

The assessment metrics measured are a set of durations depicted in Fig. 4. They were evaluated for each instantiated virtual machine. The evaluation process went through two stages. The first one consisted in quantifying each metric in order to obtain its evolution tendency and to determine the overhead introduced by VAMP in terms of execution duration. The second one focused on the user perception of different deployment durations when deploying N instances of the test application.

1) Quantification and Tendency Assessment: The measures were obtained while making the number of deployed application instances vary from 2 to 24. This corresponds to a number of virtual machines enclosed between 6 and 72 and a number of virtual machines per instantiation node ranged between 1 and 12.

Each evaluated metric can be broken down into two components. The first one is common to any application and measures the overhead introduced by the VAMP platform whereas the second one evaluates the time consumption specific to the deployed application. The distribution between these two components varies according to the considered metric. Thus, as the VAMP component instantiation and local configuration do not require any applicative processing, the local configuration duration is reduced to its VAMP specific component. Conversely the weight of this component is very limited or even negligible in both other cases. It represents indeed less than 3.3% of the remote binding duration and only 0.07% of the start duration. Consequently a quite good estimation of the VAMP time overhead consists in comparing the local configuration duration to the application specific components of the remote binding duration and of the start duration. Fig. 5 illustrates the values obtained while benchmarking the JEE application described above. This graphic also shows the values obtained for booting preliminarily the virtual machine associated.

In this benchmark, the overhead introduced by the VAMP system represents about 8% of the total duration for obtaining an operational VM. Moreover its increasing tendency (i.e. the evolution of the local configuration duration) is appreciably less marked than the start and boot duration ones.

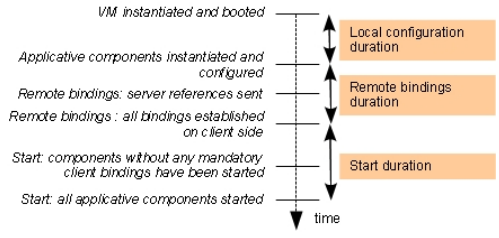

Figure 4. Assessment metrics used for benchmarking VAMP deployment mechanism and evaluating the overhead it introduces

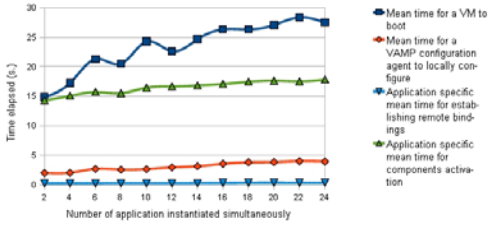

Figure 5. Evaluation of the time overhead introduced by the VAMP system

2) End-user Perception of Deployment Durations: The second step of the benchmark aims at measuring different durations that reflect the user perception when deploying simultaneously N application instances. The metrics that have been so assessed are:

- the mean time to deploy one VM (MTTD1V), i.e. the time elapsed to instantiate and to boot the VM and then to get the applicative components it embeds ready;
- the mean time to deploy one application instance (MTTD1A), i.e. the time elapsed to get ready all VMs participating to an application instance;
- the mean time to deploy N application instances (MTTDNA), i.e. the time elapsed to get the N application instances ready.

As depicted on Fig. 6, all these metrics evolve linearly towards the number of applications instantiated simultaneously. In order to evaluate the parallelism introduced by VAMP for deploying N applications, two ratios have been evaluated:

- the first one measures the average gain introduced by the deployment of one application instance with VAMP compared to deploying each of its three VMs sequentially. Its value is equals to $1 - MTTD1A/(3 * MTTD1V)$[3]. It is constant towards the number of applications instantiated simultaneously (N) and equal to 52%.
- the second one evaluates the benefit to deploy N applications with VAMP compared to deploy them sequentially. The ratio formula is $1 - MTTDNA/(N * MTTD1A)$. When making N vary from 2 to 24, it

[3] 3 is the number of VMs participating in one application instance

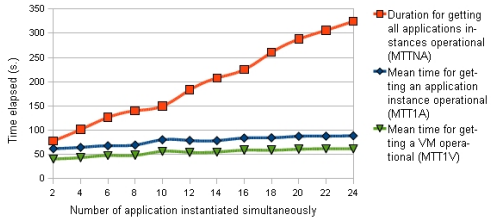

Figure 6. Duration perceived by an end-users for deploying an application with VAMP

converges to an asymptotical value where the gain is about 85%.

Both of these observations (see Fig. 7) illustrate the benefit associated to the VAMP use. Its asynchronous and decentralized self-configuration protocol allows a human administrator to reduce significantly the duration for deploying a large number of applications.

V. Related Works

The emergence of cloud computing during the last years has come with a profusion of PaaS solutions. The fragmentation of this market ensues essentially from a too general and imprecise definition of what the PaaS is. Conversely to the IaaS or the SaaS whose goal is absolutely clear (i.e. to provide the users respectively with virtualized hardware or software), according to the US National Institute of Standards and Technologies (NIST) definition, the PaaS offers models and environments to automatically manage the whole life-cycle of the applications deployed in the cloud. Such a vagueness had for consequence to delay the PaaS offers compared to the IaaS or SaaS ones.

From an industrial point of view, a lot of companies try, from now on, to enter into this promising market. Whether they are IaaS players –that wish to offer higher-level solutions based on the quite "basic" provisioning of virtualized hardware resources, e.g. [9]–, SaaS agents –that want to allow their users to customize their own business services, e.g. [10]– or newcomers –that become aware of the challenges and the economical repercussions associated to the cloud market–, each of them has as an absolute priority to conquer new PaaS market shares as fast as possible. This has for consequence a mass of heterogeneous offers towards the technological or functional spectrum they cover, the business domain they address or even their maturity. Thus [9] and [1] only deal with the deployment and elastic management of JEE enterprise web applications. Moreover [1] is only delivered as public cloud[4]. [1] imposes also a Java-like very specific programming model to which applicative code must conform (e.g. no Java threads). On its own side, the use of [9] limited to stateless applications compatible with the execution environment based on the Tomcat 6 web server. Comparably [12], that automates some deployment and elasticity aspects, is confined to the applications based on Microsoft technologies. The main restriction of [10] concerns the business domain it addresses, i.e. the customer relationship management. Finally solutions like [2] or [13] focus on a tiny part of the life-cycle management (i.e. the multi-VMs post-configuration, that consists in setting some configuration parameters when VMs boot completed). That for, such solutions adopt a declarative approach based on master/slaves mode in which each virtual machine synchronizes regularly its internal configuration with this stored on a centralized server. Nevertheless exchanging dynamic data between two slave entities (i.e. VMs) or starting an application is quite a complex task for such tools, due to an inappropriate approach based on recurrent synchronizations.

In order to address these limitations, many research works have been led. Thus, as unique standard for describing the deployment organization of a set of virtual images, OVF represents a first step toward the full and coherent formalization of a distributed application deployed in the cloud. Incidentally some key players like VMWare or Citrix already offer platforms for deploying OVF packages. In our opinion, OVF lacks a support for describing distributed architectures with their configuration, especially the dynamic configuration of the distributed bindings. The absence of such a declarative formalism implies that the configuration is either stuffed in the application code or else is executed with the help of external and ad-hoc configuration scripts [14].

[15] first discusses the implications of the architectural definition of distributed applications candidate to be deployed in the cloud. It underlines especially that such an architecture has to be reified at runtime: this is an opinion we share. Second, it proposes language elements for describing software architectures, requirements towards the underlying execution platforms, architectural constraints (e.g. concerning placement and collocation) and rules relating to applications elasticity. We plan to include in future VAMP extensions the capability to express constraints and to deal with elasticity, however the formalism presented in this article does not cover these aspects yet. Concerning the requirements description towards the underlying IaaS platforms, both approaches are based on OVF. Nevertheless they differ regarding the formalism used for describing the architecture. [15] adopts a model driven approach with extensions of the *Essential Meta-Object Facility* (*EMOF*) abstract syntax[5] whereas the current article suggests to extend an ADL. Finally, as for the deployment mechanism (protocol and architecture) -especially concerning the dis-

[4]A functionally equivalent and open-source solution is available [11] and can be deployed as private cloud

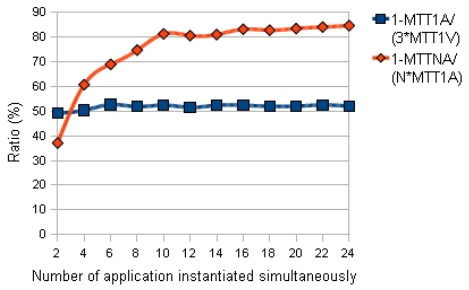

Figure 7. Evaluation of the parallelism introduced by the VAMP deployment process

[5]This syntax has been defined by the *Model Driven Architecture* (*MDA*) initiative of the *Object Management Group* (*OMG*).

tributed bindings configuration and the activation order of components that are the core of the present article-, it is not much detailed in [15].

[16] suggests an extension of *SmartFrog* [17] that enables an automated and optimized allocation of cloud resources. It is based on a declarative description of the components building up a distributed application and of the available resources. The descriptions of applicative architectures and of available resources are defined with the help of the *DADL* language. This language allows expressing, on the one hand, the applications constraints relating to the resources in terms of *Services Level Agreements* (*SLAs*) and, on the other hand, elasticity constraints. Compared to the present article, [16] focuses on the language aspects. DADL is an extension of SmartFrog, which is a Java framework for deploying distributed systems. SmartFrog is extended thanks to Java classes inheritance. The language we offer is more declarative and architecture-centric. It is based on a well known formalism for describing virtual machines (OVF) and it integrates an architecture description language (Fractal ADL). Moreover [16] does not give any details concerning the deployment process itself, on its performances or its robustness. Finally [16] intends to address the optimal resources allocation whereas the work described in this article mainly focuses on the efficiency and the reliability of the deployment process.

VI. Conclusion and Future Works

This article presents a solution for self-configuring legacy distributed applications in the cloud. A first contribution of the article is a formalism for describing a legacy application distributed on a set of virtual machines. It extends the OVF formalism, addressing virtual machines description, with an architecture description language (ADL). This extension allows specifying explicitly and in a declarative way the components building up an application and the bindings between these components. A second contribution is a dynamic and decentralized self-configuration protocol part of an automated deployment tool at Orange Labs. This decentralized self-configuration protocol is the core of the article; we argue that it can be reused for different legacy software and it improves the efficiency of the deployment process compared to a sequential approach. A third contribution is a performance evaluation when deploying a JEE enterprise web application on a private IaaS platform.

The properties of the proposed mechanism (decentralization and communications asynchronism) open up interesting horizons in terms of reliability of the deployment process. Beyond the enforcement of the deployment protocol reliability, which could for instance be implemented on a compensation-based mechanism, the future extensions of the work described in this article include (i) the reliability of the deployed applications, i.e. self-repairing and more generally other autonomic properties like self-optimization, (ii) the management of applications elasticity in the description formalism and in the engine executing the applications. Finally, the management of multi-IaaS aspects would be a pertinent extension from an industrial point of view.

References

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

[2] Puppet Labs website. http://docs.puppetlabs.com/

[3] N. Medvidovic and R. N. Taylor, "A framework for classifying and comparing architecture description languages," *IEEE Transactions on Software Engineering*, vol. 26, pp. 70–93, January 2000.

[4] E. Bruneton *et al.*, "The fractal component model and its support in java," *Softw., Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[5] K. Farooqui *et al.*, "The iso reference model for open distributed processing: An introduction," *Computer Networks and ISDN Systems*, vol. 27, no. 8, pp. 1215–1229, 1995.

[6] J. de Meer, "The iso reference model for open distributed processing," *Computer Networks and ISDN Systems*, vol. 27, no. 8, pp. 1211–1214, 1995.

[7] *Open Virtualization Format Specification*, Distributed Management Task Force DMTF Standard, Rev. 1.0.0, 2009.

[8] X. Etchevers *et al.*, "Self-configuration of Distributed Applications in the Cloud," in *4th International Conference on Cloud Computing (CLOUD 2011), Washington D.C., U.S.A., July 4-9, 2011*. IEEE, 2011, p. to be published.

[9] AWS Elastic Beanstalk website. http://aws.amazon.com/fr/elasticbeanstalk/

[10] Salesforce.com website. http://www.salesforce.com/

[11] N. Chohan *et al.*, "Appscale: Scalable and open appengine application development and deployment," in *CloudComp*, ser. Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, D. R. Avresky *et al.*, Eds., vol. 34. Springer, 2009, pp. 57–70.

[12] Microsoft Azure website. http://www.microsoft.com/windowsazure/

[13] Opscode Chef website. http://wiki.opscode.com/display/chef/Home

[14] Usharesoft Open Appliance Studio website. https://www.usharesoft.com/products/oas.html

[15] C. Chapman *et al.*, "Software architecture definition for on-demand cloud provisioning," in *HPDC*, S. Hariri and K. Keahey, Eds. ACM, 2010, pp. 61–72.

[16] J. Mirkovic *et al.*, "Dadl: Distributed application description language."

[17] P. Goldsack *et al.*, "The smartfrog configuration management framework," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 16–25, January 2009.