Self-Configuration of Distributed Applications in the Cloud

Conference Paper · July 2011

DOI: 10.1109/CLOUD.2011.65 · Source: DBLP

4 authors:

Xavier Etchevers
Orange Labs
16 PUBLICATIONS **71** CITATIONS
SEE PROFILE

Thierry Coupaye
Orange Labs
75 PUBLICATIONS **2,262** CITATIONS
SEE PROFILE

Fabienne Boyer
University Joseph Fourier - Grenoble 1
50 PUBLICATIONS **534** CITATIONS
SEE PROFILE

Noel De Palma
University Joseph Fourier - Grenoble 1
100 PUBLICATIONS **758** CITATIONS
SEE PROFILE

Self-configuration of distributed applications in the cloud

Xavier Etchevers, Thierry Coupaye
Orange Labs
Meylan, France
{*xavier.etchevers, thierry.coupaye*}*@orange-ftgroup.com*

Fabienne Boyer, Noel de Palma
University Joseph Fourier
LIG Labs
Grenoble, France
{*fabienne.boyer, noel.de_palma*}*@inrialpes.fr*

***Abstract*—In the field of cloud computing, current solutions dedicated to PaaS (Platform as a Service), i.e. the environments that deal with the different stages of the application life-cycle, remain business domain specific and are only partially automated. This limitation is due to the lack of an architectural model for describing a distributed application in terms of its software stacks (operating system, middleware, application), their instantiation as virtual machines, and their configuration interdependencies.**

This article puts forward (i) a component-based application model for defining any kind of distributed applications composed of a set of interconnected virtual machines, (ii) an automated line for deploying such a distributed application in the cloud, which includes a decentralized protocol for self-configuring the virtual application machines, (iii) a first performance evaluation demonstrating the viability of the solution.

***Keywords*-cloud computing; distributed applications; virtualization; deployment; autonomic computing;**

I. Introduction

Cloud computing environments [1] fall under three main kinds of offers according to the resources they provide. The Infrastructure as a Service (IaaS) level enables the access to virtualized hardware resources (processing, storage and network). The Software as a Service (SaaS) layer aims at providing the end-users with software applications. The intermediary layer, called Platform as a Service (PaaS), offers a set of tools and runtime environments that allow managing the applications life-cycle. This life-cycle includes the phases related to the design, the development, the deployment of applications, and generally speaking all their management stages (workload, fault tolerance, security).

This article focuses on the deployment of distributed applications in virtualized environments such as *cloud computing*. Such deployments require to generate the virtual images that will be instantiated as virtual machines, thus ensuring the execution of the application on an IaaS platform. Each image embeds technical elements (operating system, middleware pieces) and functional ones (data and applicative software entities). Once it has been instantiated, each virtual machine is subjected to a stage of dynamic settings, which finalizes the global configuration of the distributed application.

Indeed each virtual machine includes a myriad of configuration parameters relating to its entire software stack. 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). According to their nature, these parameters can be set as the image is generated or after the associated virtual machine has been instantiated. One of the main difficulties to deploy a distributed application in the cloud lies in the number of parameters to be set, their variety and the time when they can / have to be set.

On the whole, the deployment solutions currently available do not take into account these different configuration parameters, which are mostly managed by dedicated scripts. Moreover these solutions are not able to automate the images generation, their instantiation as virtual machines and their configuration independently from the kind of distributed application to be deployed. For instance, Google App Engine [2] solution only deals with Web services organized into precisely defined tiers. In our opinion, the absence of general solutions results essentially from a lack of formalism for describing the distributed application architecture with its configuration constraints in a virtualized infrastructure such as cloud computing.

This article presents a general solution named VAMP, for Virtual Applications Management Platform, that automates the deployment of any distributed applications in the cloud. The suggested approach is architectural, meaning that it is based upon an explicit representation of the applications' distributed architecture. We offer, on the one hand, a formalism for describing an application as a set of interconnected virtual machines and, on the other hand, an engine for interpreting this formalism and automating the application deployment on an IaaS platform.

Specifically, this article makes three contributions:

- A formalism that offers a global view of the application to be deployed in terms of components with the associated configuration- and interconnection constraints and with their distribution within virtual machines. This formalism extends OVF language, dedicated to virtual machines description, with an architecture description language [3] (ADL) that allows describing a distributed

application software architecture;
- A deployment engine, i.e. a runtime support able to deploy automatically an application described with this formalism. This engine is based on a decentralized protocol for self-configuring the instantiated virtual machines. In our opinion it can ease the scalability of the dynamic configuration stage;
- A performance evaluation of the proposed solution on an industrial IaaS platform (Eucalyptus [4]).

This article is organized as follows. In section II we suggest a formalism for modelling an application to be deployed in the *cloud*. Section III focuses on the line for automating the application deployment. Section IV describes the decentralized protocol for self-configuration. An evaluation of this support follows in section V. Section VI positions our contribution in relation to current works in the field of application deployment in the cloud. Section VII concludes and outlines perspectives for further research.

II. Virtualized Application Model

This section brings in the extended OVF model that is suggested for describing applications.

A. OVF Extension Principles

OVF (Open Virtualization Format) is a standard under definition provided by the Distributed Management Task Force (DMTF) [5]. It consists of a declarative, extensible and XML-based formalism that allows describing the deployment organization of several virtual images, and characterizing explicitly the virtual machines in which they will be instantiated. Thus, it is possible to set the processing-, storage- and network capabilities of each virtual machine, their start order, their requirement concerning a potential reboot after reconfiguration. OVF also makes it possible to define variables whose value will be statically or dynamically (at runtime) set by the user or by the execution environment.

The formalism we offer introduces two extensions to OVF:

- The first one is a new section (*AppArchitectureSection*) in the existing format. It aims at offering an architectural view of the distributed application contained in the OVF package. The chosen formalism for describing the applicative architecture within this new section is an ADL.
- The second one is a modification of the *References* section listing the resources referenced by the OVF package. This section includes notably the images definition used when instantiating the virtual machines. However these resources are static insofar as they must exist at the OVF package creation. Yet, the automation of the entire deployment stage of an application in the cloud, and more precisely the creation of associated software images, require to describe resources that are not yet available and that will be generated on the fly. Extending the *References* section with the *DynamicImage* element allows defining the entities that participate in image generation, i.e. operating system, middleware units and applicative components and data to be embedded.

B. Distributed Application Description

The distributed application description, as expected in *AppArchitectureSection* section of the extended OVF format, rests on the architecture description language of the Fractal component model [6] [7]. This language offers to define the application global structure in terms of components, configuration and interconnections[1]. A component includes a set of interfaces that can be either exposed (server interfaces) or required (client interfaces). A client interface of a component and a server interface of another component can be associated through a binding. Each component also contains some location related information (*virtual node* property) that designates, in the case of VAMP, the virtual machine on which the component has to be installed.

This language offers a higher abstraction level compared to configuration scripts that are specific. It allows automating the configuration tasks relating to the whole software stack within a virtual machine. Moreover it enables the modelling of legacy applications thanks to components that wrap the configuration [8].

Fig. 2 gives an example of an extended OVF descriptor for the basic *TokenApp* application described in Fig. 1. This application consists of three interconnected components, named respectively *C0*, *C1* and *C2*. Each of them owns a server interface *s* and a collection of client interfaces *c*. Each component is deployed on a distinct virtual machine. Each component is interconnected with the two others through bindings between its client interfaces and their server interfaces.

III. Automated Deployment Line

Building upon the formalism described in section II, this section presents a processing line for the self-deployment of a distributed application in the cloud.

As illustrated in Fig. 3, this processing line starts with the image generation and publication:

- **Image generation**: the first step consists in creating, from an application definition based on the formalism described previously, all the images corresponding to the virtual machines building up the application[2].
- **Image publication**: in a second time, the generated images are published within an IaaS platform. They are

[1]The Fractal model is also recursive (a component can consist of sub-components in a hierarchical way at any depth) and reflexive (the architecture is explicit and can be manipulated at runtime)

[2]The generated images offer a minimal footprint, i.e. they only include the packages they require to run.

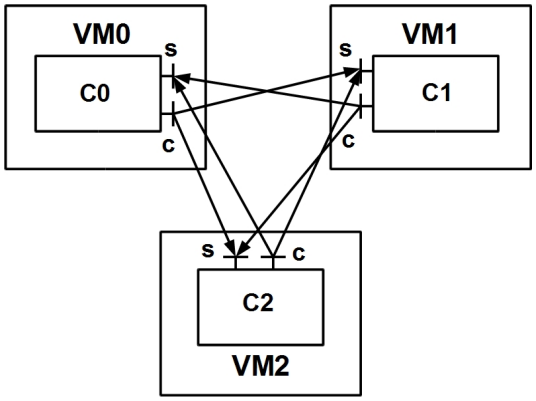

Figure 1. Example of the basic *TokenApp* distributed application

copied in the platform storage system and then made accessible through its images repository.

The image generation is carried out by an entity named *Images Manager*. Besides the software elements required for running the application (operating systems, middleware entities and application binaries), VAMP adds in each image the whole or a part of the configuration properties that are independent from the execution environment. Finally, in order to offer self-configurable virtual machines, VAMP also includes a local representation of the applicative architecture, i.e. reduced to the components and the bindings it manages, and a *configurator* in charge of interpreting this representation for instantiating, configuring and activating the applicative components for which it is responsible[3]. The image publication is then managed by a *Placement Manager* that interacts with the IaaS platform through the API it exposes. After image generation and publication, the self-deployment processing line goes on according to the stages below:

- **Image instantiation**: in this step, the published images are instantiated as virtual machines in the IaaS platform. From there on, each virtual machine starts up independently and after it has finished to boot, the embedded *configurator* is automatically instantiated. Self-configuration steps can now start.[4]
- **Local and global configuration**: each configurator conducts the dynamic configuration of the application, i.e. of all the local parameters relating to the software elements that form the virtual machine as well as of dependencies with other virtual machines.
- **Application activation**: finally, each configurator takes part in starting the application. It activates the applicative components included within the virtual machine for which it is responsible according to the associated start constraints.

[3]During the self-configuration stage, the configurator uses the local definition of the applicative architecture for determining the components it must instantiate and their needs in terms of local and global configuration.

[4]The image instantiation is gathered together with the image publication within the Placement Manager.

```
<Envelope ...>
  <References>
    <!-- Static file that contains the VAMP jar -->
    <File ovf:id="vampJar"  ovf:href="vamp.jar"/>
    <!-- A dynamic image definition -->
    <DynamicImage ovf:id="appDisk" pm:version="1.0"
                  ovf:required="false">
      <OperatingSystem pm:id="Debian" pm:version="5.0"
                       pm:architecture="i386"/>
      <Products>
        <Product pm:id="Java" pm:version="1.6"/>
        ...
      <Products>
    </DynamicImage>
  </References>
  <DiskSection>
    <Disk ovf:diskId="appDiskId" ovf:capacity="1024"
          ovf:capacityAllocationUnits="byte * 2^20"
          ovf:format="http://xen" ovf:fileRef="appDisk"/>
  </DiskSection>
  ...
  <!-- Applicative architecture -->
  <AppArchitectureSection>
    <definition name="TokenApp">
      <component name="C0">
        <interface name="c" role="client" .../>
        <interface name="s" role="server" .../>
        ...
        <virtual-node name="VM0"/>
      </component>
      <component name="C1">
        ...
      </component>
      <component name="C2">
        ...
      </component>
      <binding client="C0.c" server="C1.s" />
      <binding client="C0.c" server="C2.s" />
      ...
    </definition>
  </AppArchitectureSection>
  ...
  <!-- Virtual machines configuration -->
  <VirtualSystemCollection ovf:id="App">
    <!-- VM0 configuration -->
    <VirtualSystem ovf:id="VM0" rsrvr:min="1"
                   rsrvr:max="1">
      ...
      <VirtualHardwareSection>
        <Item>
          <rasd:ElementName>Harddisk 1</rasd:ElementName>
          <rasd:HostResource>
            ovf://disk/appDiskId
          </rasd:HostResource>
          ...
        </Item>
        ...
      </VirtualHardwareSection>
    </VirtualSystem>
    ...
  </VirtualSystemCollection>
</Envelope>
```

Figure 2. Example of an extended OVF descriptor that includes the applicative global architecture in section *AppArchitectureSection*. The elements in bold font depict the extensions introduced by VAMP

The local and global configuration allows setting up dynamic parameters whose value is only known after the as-

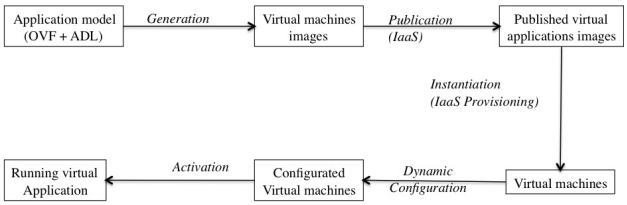

Figure 3. Description of the process for self-deploying an application in the cloud with VAMP

sociated virtual machine has been instantiated. For instance, if an IaaS platform offers a mechanism for pre-booking IP addresses that can be called upon without instantiating a virtual machine, it will be possible to set the address of a virtual machine when creating the associated image. In the opposite case, the virtual machine IP address will be known only while instantiating and VAMP will propagate it globally during the dynamic configuration stage. This step is detailed in section IV.

IV. Self-configuration and Start Protocol

The second contribution of this article is an asynchronous and decentralized mechanism for the self-configuration and activation of applicative components. Self-configuration is driven by the configurators within each virtual machine. Each of them carries out three functions:

1) based on the local architectural definition available, it instantiates the applicative components for which it is responsible and configures them locally. This consists mostly in inferring the bindings between the components deployed within the virtual machine.
2) it participates in the application's global configuration while inferring the remote bindings. A remote binding is a binding between an interface of a component that is collocated with the considered configurator in the same virtual machine and an interface of a component located in another virtual machine. Since this second component is managed by another configurator, the configurators will need to know one another before they can start communicating together. If the IaaS platform offers a mechanism for pre-booking IP addresses, this information can be stored in the virtual machine as a static file included during the image generation. Otherwise it is determined at runtime thanks to dynamic discovery based on a broadcast protocol.
3) it starts the applicative components for which it is responsible (those collocated with it within a common virtual machine) according to an order relating to the dependencies between applicative components. In VAMP, these dependencies are modelled thanks to a property named *contingency* that characterizes a client interface of a component C. It indicates if the considered client interface must be linked to a server interface, through a binding, so that component C can be activated. In such a case, the interface contingency is *mandatory*. Otherwise the contingency is qualified as *optional*. By extension, the contingency of a binding designates the contingency of its client side. Thus a component owns a mandatory / optional binding if the client side of the considered binding is one of the component interfaces.

The last two steps require communications to be established between the different configurators. These exchanges are carried out thanks to messages sent through a decentralized, asynchronous and reliable *Messages Oriented Middleware* (MOM) that is distributed on all virtual machines. Its decentralized architecture aims at removing any potentially shared resources synonymous with risks in terms of scalability, be it performances (bottlenecks) or reliability (critical resource) issues. Its asynchronism (with communications in disconnected mode) allows sending messages to a recipient that does not exist yet or that disappeared temporarily due to a failure. The MOM also offers messages persistency until they have been handled by the recipients, even if the sender or the recipients fail meanwhile. Both asynchronism and persistency improve the deployment agility insofar as each component can be started independently and thus contributes to the application processing in a degraded mode.

The stages for establishing the remote applicative bindings and then for activating the components are organized according to the following protocol:

1) For each binding λ associated to a component for which configurator γ is responsible and whose server side is local,[5] γ sends to configurator γ' responsible for λ client side a message that contains the server's side reference (*bind*). This reference includes all information required by the client component to interact with the server component.
2) When a configurator receives a message containing such a reference, this means it is responsible for the binding client side. Thus it proceeds with the binding local configuration.
3) Once it has sent all its server references, a configurator 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.
4) Thus, 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 (*start*) to each configurator responsible for the associated client side.
5) Upon receiving a start message, a configurator determines whether the component targeted by the message

[5]For simplifying, a binding side is local (respectively remote) if it is an interface of a local (respectively remote) component.

has all server references for each of its mandatory bindings. In such a case, the component is activated and the configurator executes step 4 on this component. Step by step the application is fully started.

Fig. 4 provides an example of the VAMP self-configuration and activation protocol execution for the *TokenApp* application.

V. Evaluation

The goal of this evaluation is to assess the viability of the VAMP process for deploying an application. It focuses only on the speed of deployment.

The *TokenApp* application, deployed during these tests, is formed with N identical applicative components (see Fig. 1). Each client interface of each component has an optional contingency. The graph made up with the components and their bindings has a complete grid structure. Such a structure corresponds to the most unfavorable applicative architecture for a self-configuration process. The message content for bindings establishment consists of about ten bytes, stored on the disk when received on client side. None of the applicative component is collocated with another one. Thus each of them is deployed in an own virtual machine that consists of a virtual CPU and 128 MB virtual memory. For each virtual machine, the associated image size is 1 GB.

A. Tests Environment

The validation environment that has been used is a private cloud platform. It consists of 6 Dell Studio Hybrid machines (1 x Intel Core 2 Duo T8100 2.1 GHz, RAM 4 GB and HDD 320 GB) linked through an Ethernet 100 Mbps local area network. Each machine is installed with a Linux (Debian Lenny) operating system, a Xen hypervisor (v3.2) and the Eucalyptus IaaS platform (v1.6.2) that is partially open source. One of the physical machines is used as Eucalyptus front-end. It gathers together all the IaaS management entities, i.e. the *cloud controller*, the *cluster controller*, the *storage controller* and the *Walrus* storage system. Each remaining physical server runs a *node controller* for ensuring virtual machines instantiation. Such a server is called *instantiation node* or *deployment node*. In the context of these experimentations, the images repository of the IaaS platform is centralized. However, when instantiating a virtual machine, a copy of the associated image is transferred and then stored in the target deployment node. Indeed the IaaS platform does not offer any NAS-based (Network Attached Storage) storage mechanism to keep the instantiated images copies.

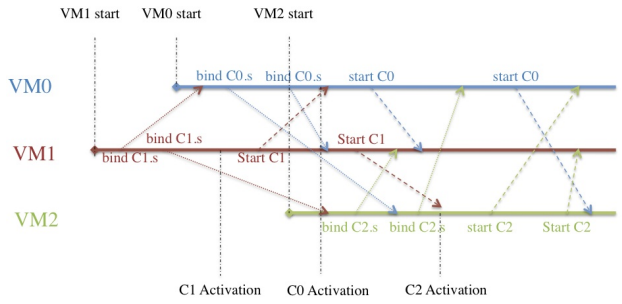

Figure 4. Example of automated configuration and activation with VAMP: the configurator embedded in VM1 sends its server interface references to the configurators with which it shares bindings. However it does not need to wait for them to be instantiated for sending them messages. Once started, it notifies the configurators that depend on it based on their contingency.

Eucalyptus implements Amazon Web Service (AWS) EC2 [9] and S3 [10] APIs. Out of the operations they provide, we used the following ones for the evaluation:

- creating an IaaS usable bundled image from an image stored on a standard file system in Xen format;
- transferring this bundled image to the IaaS storage system;
- registering a stored element in the images repository;
- instantiating a new virtual machine on a deployment node from an image registered in the repository;
- deleting a virtual machine;
- deregistering an image from the repository;
- removing an element of the IaaS storage system.

B. Results

The assessment metrics measured are a set of durations depicted in Fig. 5. They were evaluated for each instantiated virtual machine. The evaluation campaign went through three steps. The first one consisted in quantifying each metric and validating the expectations concerning its evolution tendency. The second measured the correlation between the available memory within a virtual machine and the assessed durations. Finally, the third evaluated the application deploy-

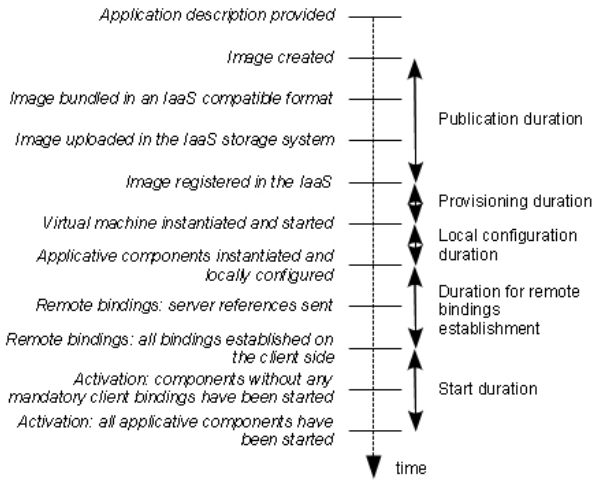

Figure 5. Assessment metrics used for evaluating the viability of the VAMP deployment mechanism

ment duration as it is perceived by the end-user.

1) Quantification and Tendency Assessment: The measures were obtained while making the number of deployed applicative components (i.e. the number of instantiated virtual machines) and the number of instantiation nodes (2 then 5) vary.

The mean value of the duration for publishing an image in the IaaS varies a lot in accordance with the number of simultaneous publications. From 115 s. per image for an isolated publication, it increases to 200 s. per image for 20 simultaneous publications. This evolution speaks for factorizing the published images which compose the application. This leads to a greater number of post-instantiation self-configuration operations.

The provisioning duration was then assessed when deploying N virtual machines. Each of them was based on a distinct image that had been first published in the IaaS. When there are more than twenty four parallel instantiations, the provisioning duration increases exponentially due to the IaaS front-end collapse. In order to keep acceptable tests conditions, we made the hypothesis that the *TokenApp* application would be reduced to a unique image that is instantiated several times (according to N value). This assumption implies that any configuration operation, which could be virtual machine specific, is not statically carried out when generating the associated image. Fig. 6 illustrates the results obtained for the provisioning duration[6]. This provisioning duration depends, on the one hand, on the number of virtual machines deployed on each instantiation node and, on the other hand, on the time for transferring images from the IaaS storage system to the instantiation nodes. These two dimensions impact respectively on the instantiation nodes overload and on the network flooding. Their effects on the results obtained are antagonistic. The blue curve (2 instantiation nodes) has a steeper slope than the red curve (5 instantiation nodes) thus demonstrating the primacy of the instantiation nodes overload over the network flooding. However, for a reduced number of virtual machines, the provisioning duration on 2 instantiation nodes is less important than on 5. In such a case, the instantiation nodes are not heavily loaded and the network flooding becomes the limiting factor: the time for transferring images through the network increases with the number of instantiation nodes.

As for Fig. 7, it shows the results for the local configuration duration as well as the duration for establishing the remote bindings. The first one is almost constant toward the total number of virtual machines whereas the latter one depends on it linearly, due to its correlation with the number of bindings a configurator establishes. Conversely, the virtual machines distribution on the different instantiation nodes has

[6]This duration, related to the underlying IaaS platform, includes the time for transferring the image from the IaaS storage system to the instantiation node and the time for the virtual machine to boot

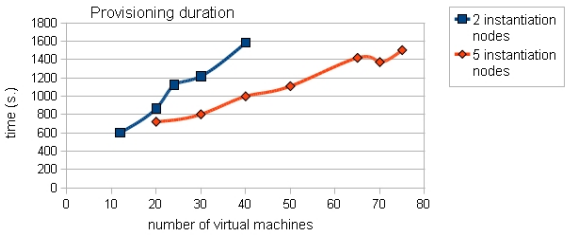

Figure 6. Average duration for provisioning a virtual machine according to the number of virtual machines deployed on two (P2) then five (P5) instantiation nodes

a stronger impact on the local configuration duration than on the duration for establishing the remote bindings.

As for start duration, its value is so insignificant compared to all other metrics (it never exceeds 0.2 s.) that its tendency was studied.

2) Virtual Memory Impact: The second step of the evaluation focuses on the relationship between the available virtual memory and the performances. Thus, during this step, the deployed virtual machines have 512 MB RAM that are four times more than the initial ones. When deploying a number of virtual machines per instantiation node low enough for the hypervisor to keep them all in memory without using any disk swap (the most favorable case for virtual machines with 512 MB of virtual memory), the assessed durations remain roughly identical (less than 5% difference).

3) Deployment Duration Perceived by the End-user: This last testing phase offers a first evaluation of the global duration for deploying an application in the cloud. It consists in measuring the experimentation time, i.e. the maximum, applied on the whole of the configurators, of the sum of the metrics presented in Fig. 5. As depicted on Fig. 8, this time evolves similarly to its main component, i.e. the provisioning duration. The time for processing the self-configuration protocol remains not significant (about 2%) compared to the provisioning duration.

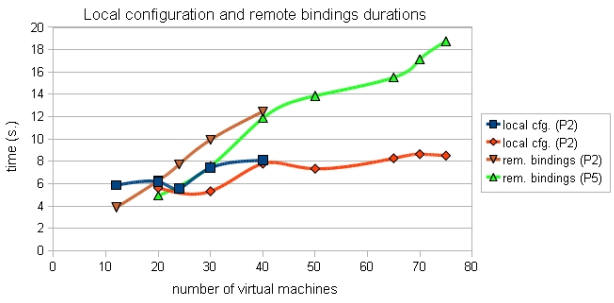

Figure 7. VAMP self-configuration and activation protocol efficiency: average durations for local configuration and for establishing the remote bindings according to the number of virtual machines deployed on two (P2) then five (P5) instantiation nodes

VI. Related Works

The formalisms and mechanisms offered by the industrial solutions for deploying applications in the cloud are generally basic, proprietary, not exhaustive and not extensible: they do not permit neither a fine-grained description of the distributed application, nor the management of its deployment process, especially its static and dynamic configuration. Moreover such solutions have often important restrictions concerning:

- the programming models like Google App Engine that only deploys web applications whose code must conform to very specific APIs (e.g. no Java threads);
- the underlying technologies like Microsoft Azure [11] that is confined to the applications based on Microsoft technologies;
- the business domains they address such as Salesforce.com [12] that focuses on customer relationship management.

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.

[13] 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. [13] adopts a model driven approach with extensions of the *Essential Meta-Object Facility* (*EMOF*) abstract syntax[7] whereas the current article suggests to extend an ADL. Finally, as for the deployment mechanism (protocol and architecture) -especially concerning the distributed bindings configuration and the activation order of components that are the core of the present article-, it is not much detailed in [13].

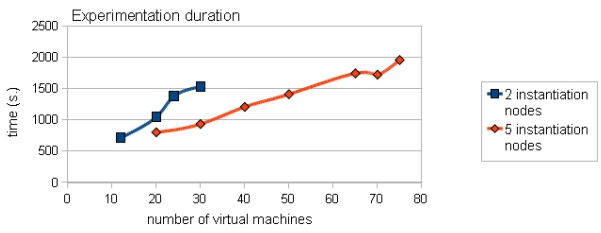

Figure 8. Time perceived by end-users for deploying an application with VAMP

[14] suggests an extension of *SmartFrog* [15] 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, [14] 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 [14] does not give any details concerning the deployment process itself, on its performances or its robustness. Finally [14] 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.

VII. Conclusion and Future Works

This article presents a solution for deploying distributed applications in the cloud. A first contribution of the article is a formalism for describing an 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 and the deployment engine that implements this protocol and that carries out the full deployment process (for some steps it uses tools that are not described in the article, especially regarding images creation and instantiation as virtual machines). This decentralized self-configuration protocol is the core of the article;

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

we argue that it improves the efficiency (scalability) and the reliability of the deployment process. A third contribution is a performance evaluation on an industrial IaaS platform. It shows the viability of the suggested solution.

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 fault recovery mechanism based on the NAS technology for storing the instantiated images, 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] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, "Above the clouds: A berkeley view of cloud computing," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2009-28, Feb 2009. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.html

[2] Google app engine website. [Online]. Available: http://code.google.com/appengine/

[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] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov, "The eucalyptus open-source cloud-computing system," in *CCGRID*, F. Cappello, C.-L. Wang, and R. Buyya, Eds. IEEE Computer Society, 2009, pp. 124–131.

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

[6] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Softw., Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[7] The fractal adl website. [Online]. Available: http://fractal.ow2.org/fractaladl/

[8] S. Sicard, F. Boyer, and N. D. Palma, "Using components for architecture-based management: the self-repair case," in *ICSE*, W. Schäfer, M. B. Dwyer, and V. Gruhn, Eds. ACM, 2008, pp. 101–110.

[9] *Amazon Elastic Compute Cloud User Guide*, Amazon Web Services, Nov. 2010. [Online]. Available: http://awsdocs.s3.amazonaws.com/EC2/latest/ec2-ug.pdf

[10] *Amazon Simple Storage Service Developer Guide*, Amazon Web Services, Mar. 2006. [Online]. Available: http://awsdocs.s3.amazonaws.com/S3/latest/s3-dg.pdf

[11] Microsoft azure website. [Online]. Available: http://www.microsoft.com/windowsazure/

[12] Salesforce.com website. [Online]. Available: http://www.salesforce.com/

[13] C. Chapman, W. Emmerich, F. G. Márquez, S. Clayman, and A. Galis, "Software architecture definition for on-demand cloud provisioning," in *HPDC*, S. Hariri and K. Keahey, Eds. ACM, 2010, pp. 61–72.

[14] J. Mirkovic, T. Faber, P. Hsieh, G. Malayandisamu, and R. Malavia, "Dadl: Distributed application description language."

[15] P. Goldsack, J. Guijarro, S. Loughran, A. Coles, A. Farrell, A. Lain, P. Murray, and P. Toft, "The smartfrog configuration management framework," *SIGOPS Oper. Syst. Rev.*, vol. 43, pp. 16–25, January 2009. [Online]. Available: http://doi.acm.org/10.1145/1496909.1496915