A Reflective Platform for Highly Adaptive Multi-Cloud Systems

Philippe Merle, Romain Rouvoy, Lionel Seinturier

A Reflective Platform for Highly Adaptive Multi-Cloud Systems

Philippe Merle, Romain Rouvoy
University Lille 1 - LIFL CNRS UMR 8022
INRIA Lille – Nord Europe
59655 Villeneuve d'Ascq, France
`firstname.lastname@inria.fr`

Lionel Seinturier
University Lille 1 - LIFL CNRS UMR 8022 & IUF
INRIA Lille – Nord Europe
59655 Villeneuve d'Ascq, France
`Lionel.Seinturier@univ-lille1.fr`

Abstract

Cloud platforms are increasingly used for hosting a broad diversity of services from traditional e-commerce applications to interactive web-based IDEs. However, we observe that the proliferation of offers by Cloud vendors raises several challenges. Developers will not only have to deploy applications for a specific Cloud, but will also have to consider migrating services from one cloud to another, and to manage applications spanning multiple Clouds. In this paper, we therefore report on a first experiment we conducted to build a multi-Cloud system on top of thirteen existing IaaS/PaaS. From this experiment, we advocate for two dimensions of adaptability— design and execution time—that applications for such systems require to exhibit. Finally, we propose a roadmap for future multi-Cloud systems.

1. Introduction

Cloud computing is a major trend in current research for distributed computing environments. Cloud computing emerged as a way for "*enabling convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction.*" [10]. Several layers of Cloud computing exist, from the infrastructure, platform, and application layers, which provide for the end-users functionalities referred to as IaaS, PaaS, and SaaS, respectively [13]. Amazon EC2, Microsoft Azure, and Google App Engine are the three most well-know Cloud platform providers, yet the offer has increased rapidly over the last months and tens of solutions are now available[1]. Besides, many big players in the IT business are also offering private Cloud solutions for their data centers.

Nonetheless, this proliferation of solutions raises several key challenges: *(i) Migration.* To avoid the vendor lock-in syndrome, services should be able to migrate, either statically or dynamically, from one Cloud provider to another. Static migration is the ability to select the target cloud before deploying a SaaS. Dynamic migration is the ability to move a running SaaS for one cloud to another. While the static migration can be considered as a portability issue, the dynamic migration raises more fundamental issues related to consistency and state preservation of running SaaS. *(ii) Interoperability.* The diversity of offers combined with Cloud services going mainstream will lead to scenarios of distributed applications whose parts are hosted on different cloud platforms, and will therefore need to interoperate and cooperate through efficient and reliable protocols. *(iii) Brokering.* The diversity of pricing strategies of Cloud solution vendors will make it relevant for third party actors to come up with offers proposing the highest level of resources for the best price. This economic optimum may vary over time, or may differ depending on the kind of resources requested. Solutions for moving swiftly between these cloud solutions in order to benefit from these best offers will then be required. *(iv) Geo-diversity.* Finally, [13] advocates that small data centers, which consume less power, may be more advantageous than large ones, and that geo-diversity tends to better match user demands. This has led to the idea that federated Cloud platforms, so-called intercloud solutions [6], are needed.

Although these challenges are not yet met and remain research directions, in this paper, we advocate reflective middleware-based solutions for adapting Cloud applications between different Cloud environments, and obtain so-called multi-Cloud systems. Based on our experience with the development of the FRASCATI SCA middleware platform (cf. Section 2), we report on a first experiment (cf. Section 3) that we conducted over eleven cloud environments: Amazon EC2, Amazon Elastic Beanstalk, BitNami, CloudBees, Cloud Foundry, DotCloud, Google App Engine, Heroku, InstaCompute, Jelastic, and OpenShift. Then, we discuss two dimensions of adaptability applied in this experiment and beyond to obtain adaptive multi-Cloud systems—*i.e.,* adaptability at design time (cf. Section 4.1), adaptability at execution time (cf. Section 4.2). We compare our contribution to the related work (cf. Section 5). Section 6 concludes this paper and presents, as a roadmap for future work in the domain of middleware for Cloud computing, some of the challenges that remain to be met.

---

[1] http://upon2020.com/2011/04/the-ever-growing-list-of-paas-companies-and-paas-projects

2. Background on FRASCATI

We consider that Cloud infrastructures require to support a wide diversity of programming technics and communication paradigms to cope with the diversity of applications that they can host. Therefore, we considered the OASIS *Service Component Architecture* (SCA) standard [3] as a suitable framework for designing and developing technology-agnostic Cloud systems. SCA provides a component-based approach and targets the heterogeneous composition of various interface definition languages (WSDL, Java, etc.), implementation technologies (Java, Spring, BPEL, C++, COBOL, C, etc.), and binding technologies (Web Services, JMS, etc.).

As illustrated in Figure 1, in SCA the basic construction blocks are software *components*, which have *services* (or provided interfaces), *references* (or required interfaces) and expose *properties*. The references and services are connected by means of *wires*. SCA specifies a hierarchical component model, which means that components can be implemented either by primitive language entities or by subcomponents. In the latter case the components are called *composites*. Both component references and services can be exposed at the composite level by means of *promotion links*. SCA is designed to be independent from programming languages, interface definition languages (IDL), communication protocols, and non-functional properties. In particular, to support interaction via different communication protocols, SCA provides the notion of *binding*. For SCA references, bindings describe the access mechanism used to invoke a service. In the case of services, the bindings describe the access mechanism that clients use to invoke the service.
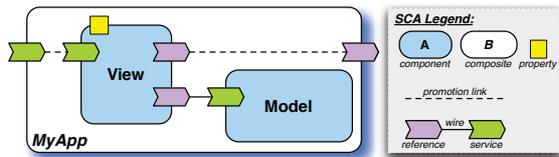

Figure 1. Overview of an SCA Component.

The code excerpt below reflects part of the configuration depicted in Figure 1 using the SCA assembly language:

```
<composite name="MyApp"
           xmlns="http://www.osoa.org/xmlns/sca/1.0">
 <service name="run" promote="View/run"/>
 <component name="View">
   <implementation.java class="app.gui.SwingGuiImpl"/>
   <service name="run">
     <interface.java interface="java.lang.Runnable"/>
   </service>
   <reference name="model" autowire="true">
     <interface.java interface="app.api.ModelService"/>
   </reference>
   <property name="orientation">landscape</property>
 </component>
 ...
</composite>
```

Listing 1. Description of the application MyApp.

We use the FRASCATI [12] middleware platform, which enables the development and the execution of distributed SOA applications based on SCA. Beyond the support of standard SCA features, FRASCATI brings reflection to SCA—*i.e.*, introspection and reconfiguration capabilities, via a specialization of the Fractal component model [5]. Figure 2 shows FRASCATI EXPLORER[2]: A reflective graphical tool provided by FRASCATI.
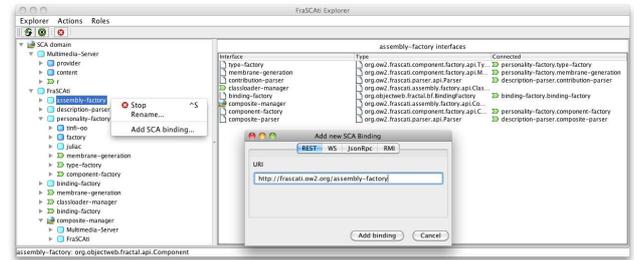

Figure 2. FraSCAti Explorer.

The FRASCATI platform is itself built as an SCA application—*i.e.*, its different subsystems are implemented as SCA components[3], which means that both the SCA applications, but also the underlying platform can be introspected and reconfiguration at execution time. Overall, FRASCATI provides a flexible and extensible component model that can be used in distributed environments to deal with heterogeneity.

3. Experiment

In order to tackle the challenges introduced in Section 1, we set up an heterogeneous multi-Cloud platform for experimenting the deployment of SaaS applications spanning different IaaS/PaaS. In this section, we therefore describe the IaaS and PaaS provisioned to build such a multi-Cloud platform, the SaaS we deployed on top of them, and finally the lessons learnt from this experiment.

3.1 Provisioned IaaS/PaaS

To build our multi-Cloud platform, we chose IaaS/PaaS providers offering free trial accounts and we used open source software. In particular, we provisioned IaaS resources (*i.e.*, CPU, memory, storage, network, load balancer, firewall, and public IP address) from *Amazon Elastic Compute Cloud* (Amazon EC2)[4] and Tata Communications's *InstaCompute*[5]. After configuring these IaaS resources, we installed a PaaS stack composed of a Linux distribution, a Java virtual machine, and a Web application container.

We also provisioned PaaS resources (*i.e.*, PaaS stacks deployed on top of provisioned IaaS resources) from *Amazon Elastic Beanstalk*[6], *BitNami*[7], *CloudBees*[8], *Cloud Foundry*[9], *DotCloud*[10], *Google*

---

[2] http://frascati.ow2.org/doc/1.4/ch07.html
[3] http://frascati.ow2.org/doc/1.4/ch12s04.html
[4] http://aws.amazon.com/ec2
[5] http://iaas.tatacommunications.com
[6] http://aws.amazon.com/elasticbeanstalk
[7] http://bitnami.org
[8] http://www.cloudbees.com
[9] http://www.cloudfoundry.com
[10] http://www.dotcloud.com
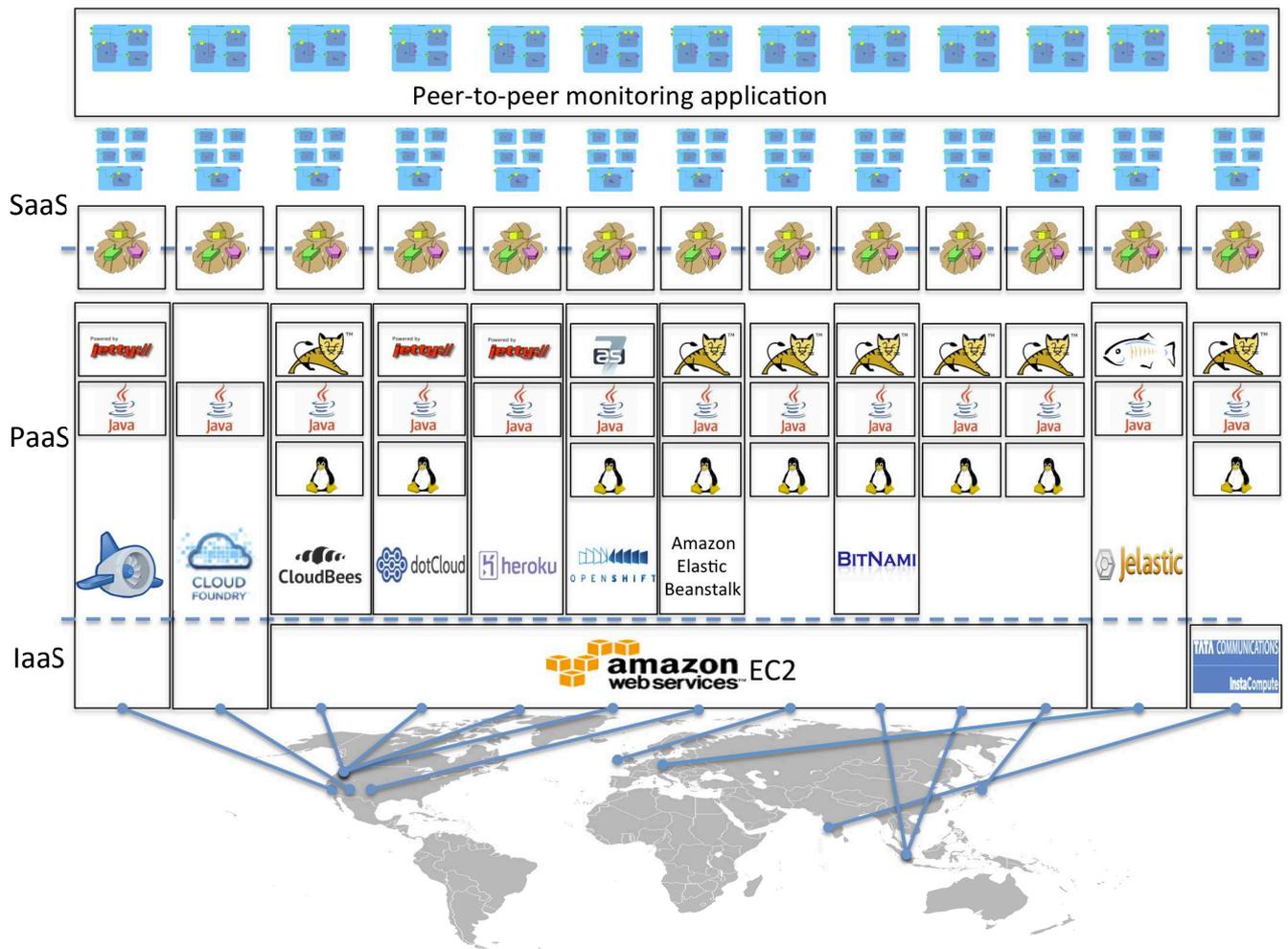
Figure 3. The FRASCATI **multi-Cloud platform.**

Cloud		IaaS hardware			PaaS software stack		
Provider	Location	CPU	RAM	Storage	OS	JRE	Web container
Amazon EC2	Asia Pacific (Singapore)	1 EC2 virtual core	613MB	8GB	Amazon Linux	OpenJDK 1.9.1	Apache Tomcat 7.0.20
Amazon EC2	Asia Pacific (Tokyo)	1 EC2 virtual core	613MB	8GB	Amazon Linux	OpenJDK 1.9.1	Apache Tomcat 7.0.20
Amazon EC2	EU North (Ireland)	1 EC2 virtual core	613MB	8GB	Amazon Linux	OpenJDK 1.9.1	Apache Tomcat 7.0.20
Amazon Elastic Beanstalk	US East (Virginia)	1 EC2 virtual core	613MB	8GB	Amazon Linux	OpenJDK 1.9.1	Apache Tomcat 7
BitNami	Amazon EC2 Singapore	1 EC2 virtual core	613MB	3GB	Amazon Linux	JVM 6	Apache Tomcat 6.0.29
CloudBees	Amazon EC2 US West	Data not available				JVM 6	Apache Tomcat 6.0.32
Cloud Foundry	US Utah	Data not available					
DotCloud	Amazon EC2 US West	Data not available				Oracle JRE 1.6.0 24-b07	Jetty 6.1.22
Google App Engine	US	Data not available					Google Jetty
Heroku	Amazon EC2 US West	Data not available				JVM 6	Jetty 7.4.5
InstaCompute	India	1 core 1.00 GHz	1GB	20GB	CentOS 5.4 X64	Oracle JRE 1.6.0 26-b03	Apache Tomcat 7.0.20
Jelastic	Germany	Data not available				JVM 6	GlassFish 3.1.1
OpenShift from Red Hat	Amazon EC2 US West	Data not available					JBoss AS 7.0

Figure 4. The provisioned multi-Cloud platform

App Engine (GAE)[11], *Heroku*[12], *Jelastic*[13], and Red Hat's *Open-Shift*[14]. Let us note that Amazon Elastic Beanstalk, BitNami, Cloud-Bees, DotCloud, Heroku, and OpenShift provision IaaS resources from Amazon EC2. All these PaaS provide a Linux distribution, a Java virtual machine, and a Web application container off-the-shelf.

Table 4 summarizes the characteristics of the thirteen nodes we provisioned for our multi-Cloud platform. The reader may have noticed that, with CloudBees, Cloud Foundry, DotCloud, GAE, Heroku, Jelastic, and OpenShift, descriptions of IaaS resources are not available because these PaaS provision them automatically and hide them to the end-user. The resulting multi-Cloud platform therefore exhibits a wide diversity of geolocations (Germany, India, Ireland, Japan, US, Singapore) and SaaS containers (Glass-Fish, Jetty, Tomcat, JBoss) as depicted in Figure 3.

3.2   Deployed SaaS

Then, we deploy FRASCATI[15], our reflective middleware platform, as a SaaS over these thirteen PaaS. But, FRASCATI can also be considered as a PaaS hosting SCA-based applications, so we develop five SaaS based on SCA: 1) the computation of Fibonacci series exposed as a Web service and a REST resource, 2) a web application for checking ISBN numbers[16], 3) a web application for checking email addresses[17], 4) a Web application for getting weather information[18], and 5) a public proxy to the secured Ohloh REST API for obtaining information about open source projects[19]. Each of these applications is implemented as an SCA composite containing one SCA component. Then, these five SOA applications are replicated on the thirteen nodes of the multi-Cloud platform as shown in Figure 3.

To experiment distributed multi-Cloud applications, we develop a peer-to-peer network monitoring application where a peer is deployed on each node of our multi-Cloud platform. Each peer is implemented by three SCA components: a *sensor* exposes local monitoring data (peer name, url, geolocation, hostname, IP address, current date, available processors, free/total/max memory) as a REST resource, an *aggregator* collects monitoring data for all peers and computes network latency, and the *view* component produces a dynamic HTML page showing a Google Map geolocating the thirteen peers, network latencies between peers, and all collected monitoring data (see Figure 5). To summarize, this monitoring application is composed of thirteen SCA composites, thirty nine SCA components, and uses two external services computing peer geolocation[20] and maps, respectively.

Orthogonally, the deployed FRASCATI includes reflective capabilities remotely accessible: 1) *remote REST management* to expose all SCA concepts presented in Section 2 as REST resources, which are introspectable and reconfigurable via HTTP requests,

[11]http://code.google.com/appengine

[12]http://www.heroku.com

[13]http://jelastic.com

[14]http://openshift.redhat.com

[15]http://frascati.ow2.org

[16]http://webservices.daehosting.com/services/isbnservice.wso

[17]http://ws.xwebservices.com/XWebEmailValidation/V2/XWebEmail Validation.wsdl

[18]http://www.webservicex.net/globalweather.asmx

[19]http://www.ohloh.net/api

[20]http://freegeoip.net

Figure 5. The multi-Cloud FraSCAti peer-to-peer network.

2) FRASCATI SCRIPT to execute complex introspection and reconfiguration scripts sent via HTTP requests, and 3) FRASCATI WEB EXPLORER a Web application to introspect and reconfigure SCA applications via dynamically generated HTML pages. Then, these reflective features provide a fine-grained management facility to deploy/undeploy SCA composites, add/start/stop/remove SCA components, get/set SCA properties, wire/unwire/bind/unbind both SCA services and references, update binding attributes, add/update/remove SCA non-functional properties, invoke service operations.

These six SOA applications plus all FRASCATI required features were packaged as a Web ARchive (WAR) of a size of 15.5MB. On the CloudBees PaaS, we observed a memory consumption of 95MB including the Java virtual machine, the Web application container, FRASCATI and the six SCA applications.

Our multi-Cloud platform is freely available at `http://fra scati.ow2.org`.

3.3   Lessons Learnt

Regarding the challenges introduced in Section 1, this experiment clearly and fully addresses *static migration* (the portability of SaaS to various PaaS/IaaS), *interoperability* (the distribution of SaaS across several PaaS/IaaS), and *geo-diversity* (SaaS/PaaS/IaaS hosted over the world) thanks to FRASCATI and SCA. As this early stage experiment started in July 2011, both *dynamic migration* and *brokering* challenges have not been addressed yet. However, the deployed peer-to-peer network monitoring application could be seen as a preliminary support for collecting static and dynamic information required to our future cloud brokering algorithms.

Some lessons can be learnt from this experiment:

Heterogeneous IaaS/PaaS management Currently, each IaaS/PaaS provides its own management tools (client SDK, Web Service or REST API, Web-based dashboard) for provisioning IaaS/PaaS resources, deploying SaaS, and monitoring all the hardware and software resources. Then for our experi-

ment, we develop different scripts for managing each targeted IaaS/PaaS. Future multi-Cloud platforms will therefore require to provide an abstraction to deal with the management variability of IaaS/PaaS. Here, SCA and FRASCATI could be appropriate candidates for designing and implementing such a multi-Cloud management facility as a service.

SaaS portability Our experiment shows the capability to deploy the same SaaS (here six SCA applications running on FraS-CAti) over different PaaS/IaaS. This SaaS portability is mainly due to the adoption of the Java programming language and the WAR packaging format. However, GAE provides a Java sandbox supporting a strict subset of standard Java API. For example, a GAE application cannot write on the filesystem, spawn a sub-process or a thread, or use a signed JAR. Thus, this requires to be able to select the right SaaS features according to IaaS/PaaS restrictions. Moreover, each PaaS/IaaS provides its own specific cloud services for user management, data storage, efficient intra-Cloud communication and coordination, etc. Another challenge will be then to provide cross-Cloud application programming interfaces that abstract common cloud platform services and make SaaS more portable.

FRASCATI as a SaaS and a PaaS FRASCATI is part of the SaaS stack (the WAR) we deploy on the thirteen Web application containers. However, FRASCATI could also be seen as a PaaS hosting the six deployed SCA-based applications. In future work, we will experiment the deployment of FRAS-CATI on top of IaaS without requiring a standalone Web application PaaS like Jetty, Tomcat, GlassFish or JBoss, *i.e.*, see FRASCATI as a native PaaS. This will reduce both the memory consumption and deployment time.

Fine-grained reflectivity and scalability FRASCATI adds an extra layer of indirection between applications and underlying PaaS/IaaS. This high level abstraction layer mainly implements SCA heterogeneity support and fine-grained reflective capabilities on top of the low level Java Servlet API provided by underlying used PaaS. Our experiment shows that SCA heterogeneity support and FRASCATI fine-grained reflectivity are compatible with Cloud computing scalability and can be combined to build large-scale heterogeneous multi-Cloud systems.

Reconfiguration and security Currently, FRASCATI provides remote fine-grained reconfiguration services but without security access control. So any remote program could reconfigure SCA applications. At the beginning of the experiment, Web robots like Google indexed HTML pages dynamically generated by FRASCATI WEB EXPLORER and then randomly start and stop SCA components of the six deployed applications introducing some denial of services. This issue was easily resolved by using the Robots Exclusion Protocol[21]. Therefore, when building public reflective cloud systems, a tension exists between fine-grained SaaS reconfiguration and security access control. A future challenge will be to mitigate this tension, especially for multi-tenants PaaS,

i.e., PaaS shared by applications deployed by different organizations. Then, each application would define its own security access control policy for fine-grained reconfiguration.

4. Different Degrees of Adaptation

We believe that the above described experiment provides a significant example of the variety of requirements that a modern distributed application has to face. The middleware system and the application have to be flexible enough in order to incorporate smoothly these different features. In the following subsections, we advocate that, to achieve such a challenge, several degrees of adaptation are required: at design time and at runtime.

4.1 Adaptation at Design Time

At design time, the adaptability of the platform relies on the combination of two mechanisms: a plugin-like SCA-based architecture, and feature diagrams.

The FRASCATI platform has been designed as a plugin-based architecture in order to let it to be adapted to many different variants of execution environments. The motivating idea is to enable a platform developer to select the right and minimal set of features she needs to compose her platform on demand. The variability obtained here mainly corresponds to the independence properties of SCA enumerated in Section 2: component implementation languages, binding technologies, and interface definition languages.

As a matter of example, 18 FRASCATI plugins exist to support different component implementation languages: Java, BPEL, Spring, Fractal ADL, OSGi (Equinox, Felix, or Knopflerfish), Scala, scripting languages (BeanShell, FScript, Groovy, JavaScript, JRuby, Jython, Xquery), Apache Velocity, Web resources. Similarly, 11 FRASCATI plugins support different binding technologies: HTTP, JMS, JSON-RPC, REST, Java RMI, SOAP, JGroups, SLP, UPnP, OSGi, and JNA. Additional FRASCATI plugins exist to support different interface definition languages (Java, WSDL, UPnP service description). These plugins are designed and implemented as SCA components, which can be embedded or not in the architecture of the platform. An API is provided for developing new plugins in order to address unforeseen requirements. Furthermore, several other functionalities of the platform (remote REST management, Web Explorer, dynamic code generation and compilation, SCA metamodel parsing, etc.) are also plugin-based leading to a set of 62 existing plugins in the FRASCATI code base[22] at the date of the writing of this paper.

The experiment described in Section 3 used 22 FRASCATI plugins: 2 component implementation languages (Java and Apache Velocity), 3 binding technologies (HTTP, REST, SOAP), 2 interface languages (WSDL, Java), 3 metamodels (OSOA SCA, FRAS-CATI SCA, FRASCATI Web), 8 core plugins (SCA parser, Assembly Factory, Component Factory, Binding Factory, remote REST management, FRASCATI Web Explorer), and 4 plugins for dynamic code generation and compilation. However, the last four plugins were not deployed on GAE because they require to write on the filesystem which is forbidden by GAE.

This plugin-based mechanism enables a first degree of flexibility on a per needed functionality basis. All the possible configurations of the platform which can of be obtained from 62 FRASCATI

---

[21] http://www.robotstxt.org

[22] http://websvn.ow2.org/listing.php?repname=frascati

plugins is huge. Yet, it appears that not all plugins are compatible with each other and that some plugins correspond to subfeatures of a given feature. In order to rule the complexity generated by this high degree of variability, in [1], we proposed *feature diagrams* to obtain a *Software Product Line* (SPL) for the FRASCATI platform. An SPL can be defined as *"a set of software-intensive systems that share a common, managed set of features and that are developed from a common set of core assets in a prescribed way"* [7]. A Feature Model is used to compactly define all features in an SPL and their valid combinations [2]. This SPL enables the developer to finely select the configuration of the platform that matches SaaS needs with the guarantee that it will be legal with respect to the composability of the FRASCATI plugins. An exhaustive description of the FRASCATI SPL is available at `http://frascati.ow2.org/doc/1.4/ch12.html`.

Overall, the plugin-based architecture and the SPL provide a way to address the interoperability and the geo-diversity challenges identified in Section 1. We can select and configure the right solutions enabling multi-Cloud systems to interoperate at the SaaS level. In addition, we believe that the SPL approach can be applied to the IaaS/ PaaS levels as well, in order to reason on the characteristics needed for deploying a particular SaaS.

4.2   Adaptation at Execution Time

The execution time adaptation of the FRASCATI platform relies on two core features: introspection and reconfiguration on one side, and dynamic deployment of reconfiguration scripts on the other side.

The FRASCATI platform follows the principles of component-based software development. The application and the platform are built as hierarchies of components. The architecture of these components are reified at runtime enabling to discover their organisation in hierarchies, their connections, the services they provide and require, their properties. Furthermore, these characteristics can be modified in order to adapt the application to new needs at runtime. Overall, this introspection and this reconfiguration can be applied at three levels: for the application executed by the platform, for the non-functional services (such as transaction, security, etc.) used by the application, and for the container hosting the application components, read [12] for more details. Currently, our multi-Cloud platform exposes both remote REST management and FRASCATI Web Explorer allowing users to introspect and reconfigure all the components of the six SCA-based applications and the FRASCATI platform.

The second feature for execution time adaptation is related to the possibility of dynamically deploying a reconfiguration script. The FRASCATI Script DSL has been designed for that [12]. FRASCATI Script provides a dedicated syntax to wrap the basic reconfiguration actions, such as navigating the component hierarchy, introspecting/modifying connections, starting/stopping components, setting properties. A dedicated API is provided by FRASCATI for remotely deploying and executing these scripts. They can, by this way, adapt the platform and the application to the new configuration. For instance, FRASCATI Script can allow to program management scripts for the monitoring peer-to-peer network, *e.g.*, adding a new peer or removing a broken down peer implies binding/unbinding all other peers, respectively.

The adaptation performed by a FRASCATI Script is currently applied at the granularity of a component or a composition of components. This is somewhat much finer than the feature level, which

has been presented in the previous section. Features are implemented by one or several components. Furthermore, their logic is closer to the level where the developer may wish to interact with the system: this is easier to reason in terms of added or removed features than in terms of the set of components which implement them. Thus, in future work, we plan to extend FRASCATI Script with feature-based reconfiguration capabilities.

5.   Related Work

Virtual IaaS solutions.

BitNami, CloudBees, DotCloud, Heroku, and OpenShift are Cloud computing providers acting as Cloud intermediaries by providing developers with an application server provisioned on top of the Amazon EC2 IaaS. These providers are therefore offering a dedicated PaaS on top of an existing IaaS, but these solutions do not cross the boundaries of Amazon EC2 and cannot be used to deploy a multi-Cloud system. The solution we propose in this paper rather builds on virtual IaaS platforms to provide a unified open programming model for SaaS, based on the SCA standard.

Provisioning of heterogeneous IaaS.

Most of the current research activities focus on the provisioning of web applications in heterogeneous Clouds [8, 9]. These approaches are resource allocation algorithms that match the resources of the node provisioned by the IaaS (CPU, I/O) in order to maximize the performances of the deployed PaaS. Although these approaches do not apply to multi-Cloud systems, we believe that they can be used for considering the automatic provisioning of Clouds according to pricing and latency dimensions. Several open source libraries exist to manage heterogeneous IaaS, i.e., deploy images and create/start/stop/destroy virtual machines, such as Apache Deltacloud[23], Apache Libcloud[24], jclouds[25], and SimpleCloud[26]. Each of them provides its own API abstracting the heterogeneity between underlying IaaS management API. Nevertheless, these abstractions are too low level as they provide an imperative programming model instead of a declarative model.

SCA-based SaaS.

To avoid the vendor lock-in syndrome, [11] proposes an SCA-based format for packaging and deploying multi-tenant aware configurable composite SaaS applications. This format extends SCA with variability descriptors and SaaS multi-tenancy patterns. We think that this SCA-based format can be reused or extended for multi-Cloud systems. To avoid lock-in to a specific cloud application platform, Apache Nuvem[27] defines an open SCA-based application programming interface for common cloud application services, allowing applications to be easily ported across the most popular cloud platforms. [4] shows how to build and integrate SCA-based composite applications using Apache Tuscany, the Eucalyptus open source cloud framework, and OpenVPN to create a hybrid composite application. Technical talks at JavaOne 2010[28]

---

[23]http://incubator.apache.org/deltacloud
[24]http://libcloud.apache.org
[25]http://www.jclouds.org
[26]http://simplecloud.org
[27]http://incubator.apache.org/nuvem
[28]http://www.slideshare.net/luckbr1975/s314011-developing-

and ApacheCon NA 2010[29] presented how to develop composite applications for the Cloud using Apache Tuscany. Our experiment is distributed across a larger multi-Cloud platform and FRASCATI brings reflective capabilities to SCA, which is not the case for Apache Tuscany.

6. Conclusion, Challenges & Perspectives

This paper reports on the first experiment towards an adaptive and reflective middleware platform for world-wide scalable heterogeneous multi-Cloud systems. We advocate that reflective middleware-based solutions are needed to adapt applications to different cloud systems. In this paper, we present the lessons learnt from deploying cloud applications on Amazon EC2, Amazon Elastic Beanstalk, BitNami, CloudBees, Cloud Foundry, Dot-Cloud, Google App Engine, Heroku, InstaCompute, Jelastic, and OpenShift. We advocate that adaptation is required both at design time and at runtime to match the broad range of variability induced by cloud platforms.

Beyond the traditional challenges identified by Cloud computing environments (scalability, elasticity, etc.), our experiments raises more specific challenges related to the development of a new generation of multi-Cloud systems. In particular, we consider that these complex systems are characterized by a wide diversity of technologies, capabilities, and locations, which requires new tooling solutions to cope with the scale and the complexity issues:

Complex Architecture Description addresses a design challenge related to the description of SaaS that spawns billions of nodes. In such a context, traditional *architecture description languages* (ADL) offer a limited support to leverage the description of these systems as they provide atomic idioms to describe exhaustively every single component involved. With regard to this challenge, we believe that high-level design approaches like design patterns or macro-programming can contribute to foster the development of complex and scalable architectures.

Consistent Software Configuration refers to the exploration of approaches, such as *software product lines* (SPL), in the context of Cloud Computing to cover the configuration of SaaS and their dependencies according to specific Cloud offers. Given that each solution imposes constraints like *software development kit* (SDK), libraries (*e.g.*, Java), operating system, or resources (memory, processor, etc.), applications have to check their configuration with regards to these constraints. We therefore consider that appropriate models should be defined to assist in the consistent configuration of the SaaS/PaaS/IaaS layers.

Continuous Service Delivery refers to the scalable and dynamic brokering and deployment of a system across a cloud of clouds. This challenge exhibits the need for an abstraction that federates existing clouds as an open infrastructure used to deploy SaaS in an agile and scalable manner. Agility refers to the capability of the infrastructure to continuously deliver SaaS whenever evolutions are required, while scalability addresses the provision and the deployment of the SaaS on a very-large number of nodes. Such a continuous service delivery could be implemented by a virtual PaaS solution that supports the seamless deployment of SaaS.

Autonomous Management copes with the challenges addressed by the autonomic computing community—*i.e.*, the monitoring, analysis, and reconfiguration of very-large-scale systems. While autonomic computing technics are already employed at the scale of a Cloud to ensure the reliability and the optimization of the infrastructure, no solution addresses the autonomous management of a SaaS deployed as a multi-Cloud system. For example, such a management infrastructure can be used to monitor the pricing variations of a given Cloud solution, and consider alternative deployments for a given SaaS. In the context of GreenIT, a green policy could consist in migrating the SaaS to a different location according to the availability of green energies (solar, wind, etc.). We therefore believe that the autonomous management infrastructure should be designed and deployed as a reflective SaaS that can scale in synchrony with the managed multi-Cloud system.

Cloud Language Unity finally groups all the above challenges into the definition of a multi-view dynamic language, which is continuously synchronized with the SaaS, independently of its current state (designed, configured, deployed, running). Although each step of the SaaS lifecycle is handled by different stakeholders, we believe that a common language should be adopted to ease the mapping and the synchronization of the concepts involved in each phase. This language should therefore be able to switch seamlessly between models used at design-time and those maintained at runtime (*e.g.*, Model@runtime) and expose the appropriate operations.

7. References

[1] M. Acher, A. Cleeve, P. Collet, P. Merle, L. Duchien, and P. Lahire. Reverse Engineering Architectural Feature Models. In *Proceedings of 5th European Conference of Software Architecture (ECSA'11)*, Sept. 2011.

[2] D. Batory. Feature models, grammars, and propositional formulas. In *Proceedings of SPLC'05*, volume 3714 of *LNCS*, pages 7–20, 2005.

[3] Beisiegel, M. et al. Service Component Architecture, Nov. 2007. http://www.osoa.org.

[4] R. Bhose and K. C. Nair. Integrating Composite Applications on the Cloud Using SCA, Mar. 2010. Dr. Dobb's, available at http://drdobbs.com/cpp/223800269.

[5] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The Fractal Component Model and its Support in Java. *Software Practice and Experience (SPE)*, 36(11-12):1257–1284, 2006.

[6] R. Buyya, R. Ranjan, and R. Calheiros. InterCloud: Scaling of Applications across multiple Cloud Computing Environments. In *Proceedings of the 10th International Conference on Algorithms and Architectures for Parallel*

---

composite-applications-for-the-cloud-with-apache-tuscany
[29]http://www.slideshare.net/jsdelfino/apachecon-na-2010-sca-reaches-the-cloud-developing-composite-applications-for-the-cloud-with-apache-tuscany

Processing (ICA3PP'10), volume 6081 of *LNCS*, pages 13–31, May 2010.

[7] P. Clements and L. Northrop. *Software Product Lines Practices and Patterns*. Addison-Wesley, 2002.

[8] J. Dejun, G. Pierre, and C.-H. Chi. Resource Provisioning of Web Applications in Heterogeneous Clouds. In *Proceedings of the 2nd USENIX conference on Web Application development (WebApps'11)*, Berkeley, CA, USA, June 2011. USENIX Association.

[9] G. Lee, B.-G. Chun, and R. H. Katz. Heterogeneity-Aware Resource Allocation and Scheduling in the Cloud. In *Proceedings of the 3rd USENIX Workshop on Hot Topics in Cloud Computing (HotCloud'11)*, Berkeley, CA, USA, June 2011. USENIX Association.

[10] P. Mell and T. Grance. The NIST Definition of Cloud Computing. Technical report, National Institute of Standards and Technology, 2009.
`http://www.nist.gov/itl/cloud/upload/`
`cloud-def-v15.pdf`.

[11] R. Mietzner, F. Leymann, and M. P. Papazoglou. Defining Composite Configurable SaaS Application Packages Using SCA, Variability Descriptors and Multi-tenancy Patterns. In *International Conference on Internet and Web Applications and Services*, pages 156–161, Los Alamitos, CA, USA, 2008. IEEE Computer Society.

[12] L. Seinturier, P. Merle, R. Rouvoy, D. Romero, V. Schiavoni, and J.-B. Stefani. A Component-Based Middleware Platform for Reconfigurable Service-Oriented Architectures. *Software: Practise and Experience (SPE)*, 2011.

[13] Q. Zhang, L. Cheng, and R. Boutaba. Cloud Computing: State-of-the-art and Research Challenges. *Journal of Internet Services and Applications*, 1(1):7–18, May 2010.