2013 IEEE 27th International Conference on Advanced Information Networking and Applications

CloudServ: PaaS resources provisioning for service-based applications

Sami Yangui and Samir Tata
Institut Mines-TELECOM, TELECOM SudParis, UMR CNRS Samovar, Evry, France
{*Sami.Yangui, Samir.Tata*}*@it-sudparis.eu*

***Abstract*—Cloud Computing involves typically provisioning of virtualized and often dynamically scalable resources for IT services operating. Study of developed Cloud platforms shows that they present limitation related to deployment and running of service-based applications that are built from heterogeneous services (programming languages, communication protocols and/or hosting frameworks). In this paper, we present CloudServ, a Platform as-a-Service (PaaS), that is dedicated to service-based applications. Its model extends the Open Cloud Computing Interface (OCCI) specification that is mainly dedicated to IaaS resources. In CloudServ, PaaS resources are continuously provisioned along with the arrival of deployment requests of service-based applications. Experiments of CloudServ demonstrate its good behaviour and highlights its scalability for a huge number of deployed services in Cloud context comparing to classical Cloud platforms.**

***Keywords*-OCCI; SCA; Service micro-container;**

I. Introduction

Nowadays, more and more companies are using Web services or even adopt new economic models based on new concepts such as *Enterprise 2.0* to adapt to new demands and expectations [1]. These new paradigms and information processing rely heavily on the use of and interaction between services. To make their services online and exploit them, companies can set up their own infrastructure or can adopt the new economic model offered by Cloud Computing.

Cloud Computing is a specialized distributed computing paradigm. It differs from traditional ones on the fact that it (1) is massively scalable, (2) can be encapsulated as an abstract entity that delivers different levels of services to customers outside the Cloud, (3) is driven by economies of scale, (4) can be dynamically configured (via virtualization or other approaches) and (5) can be delivered on demand [2].

As a part of our work, we are interested in application deployment and provisioning of platform resources in the Cloud mainly for service-based applications. Such applications consists in assembling a set of elementary services using appropriate service composition specifications like Service Component Architecture (SCA) [3]. SCA model can be defined as a pattern for building and deploying software applications. It defines and represents the structure and operation of the software application components and interaction between them. Figure 1 illustrates *Loan approval* application described using SCA. This assembly view illustrates how components reference services offered by other components which can be not implemented using the same programming languages and/or not supports the same communication protocols. For example, *HandleLoanRequest* component has been implemented as a remote EJB communicating with RMI; while *LoanApproval Process* component has been implemented as a business process (built using WS-BPEL) communicating with SOAP over HTTP protocol and *ComputeCreditRates* is a remote program written in C.

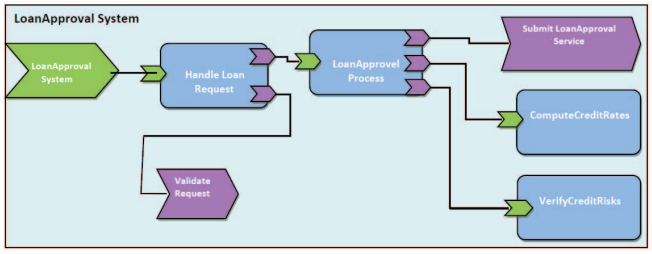

Figure 1. Loan approval service-based application described using SCA.

On the one hand, service-based applications are built from components and services that are heterogeneous in the sense that they (1) are not all implemented using the same programming languages (C++, Java, etc.), (2) do not support all the same communication protocols (RMI, SOAP/HTTP, etc.) and/or (3) do not run on the same hosting frameworks (POJO VM, .NET framework, component-based platform, etc.). These three types of heterogeneity make the deployment task of service-based applications hard to realize in Cloud environments. On the other hand, study of the state of the art (see Section II) shows that using existing Cloud platforms requires the use of specific programming languages, communication protocols and/or hosting frameworks. For example, Force.com imposes Apex language REST API [4], Azure imposes Microsoft.Net hosting framework [6], Google App Engine imposes MapReduce programming framework [5] [7], and so on. In fact, Cloud providers do not provide all the needed facilities to support the heterogeneity of service-based applications.

In this paper, we propose CloudServ, a novel PaaS, that allows developers to deploy, host and run service-based applications regardless of the heterogeneity of their programming languages, communication protocols and/or hosting frameworks. It represents an extension of the Open Cloud Computing Interface (OCCI) model [23], for the specification of platform resources (their types and relation-

1550-445X/13 $26.00 © 2013 IEEE
DOI 10.1109/AINA.2013.98

ships between them). Broadly speaking, CloudServ allows the provisioning of required platform resources to process (un)deployment service-based application requests.
The remainder of the paper is organized as follows: Section II presents the state of the art of Cloud platforms and highlights their limits regarding service-based application deployment. Section III describes CloudServ resources and provisioning model. Section IV presents how CloudServ resources are provisioned in a dedecated way for service and application deployments. In Section V, we detail the experimentations for validating our proposed platform. Finally, we conclude our paper and present our future work in Section VI.

II. State of the art

In this section, we present a state of the art of different classical Platform as-a-Service (PaaS) architectures. Then, we dicuss some related works which are interested in environment management and service deployment in the Cloud. Finally, we analyse these platforms regarding our targeted requirements *i.e.* supporting heterogeneity of programming languages, communication protocol and/or hosting frameworks.

A. Classical PaaS architectures

1) Cloud Foundry: Cloud Foundry [10] is a VMWare open source Cloud Computing PaaS software written in Ruby [16]. Among Cloud Foundry's components, we quote the *router*, the entry gate to Cloud Foundry PaaS, that routes requests to the correspondent *Droplet Execution Agent* (DEA) that runs the appropriate service. DEAs contains service containers and other engines that are necessary to run deployed services. For example, to host and run a Java service, the DEA will use an embedded Apache Tomcat.

2) CloudBees: CloudBees is an open source PaaS for Java Web applications [11]. CloudBees offers two services *DEV@Cloud* and *RUN@Cloud*. *DEV@Cloud* provides a development, building and testing environment. It allows users to develop, test and deploy services continuously with Jenkins integration server [14]. *RUN@Cloud* is modular, as these modules contain application servers (*e.g.* Apache Tomcat for J2EE) for hosting deployed applications and Java Virtual Machines (JVM) for their execution.

3) OpenShift: OpenShift is the Red Hat open source PaaS solution. OpenShift provides several APIs for developers and offers support for different languages [15]. There is a set of back-end services for management and configuration in the PaaS [12]. Currently, OpenShift can only run on the Amazon Web Services (AWS) platform provided by Amazon.com [17].

4) Jelastic: Jelastic is a Java hosting platform that runs and scales up any Java application. The PaaS offers four service containers:. Apache Tomcat, Jetty, JBoss and GlassFish for Java service hosting through an interactive Web interface. Jelastic uses AWS platform.

B. Other approaches

At the best of our knowledge, there is not yet much works that focused specifically on limitation constraints imposed by classical Cloud providers and platforms. However, there is some works who treated this problem in Grid environments [19] and even at Infrastructure layer for Clouds environments [20]. For IaaS layer, there is some deployment engine extensions that have been developed to orchestrate IaaS service deployement taking into account the service environment *e.g. Capistrano* [21] and *Control Tier* [22]. In [18], the authors designed a resource management system to adress heterogeneity resource co-allocation problems. The resource management system is introduced as a function of the Dynamic Service Deployment (DSD) module. To deploy a service, DSD install its code in a processed resource according matchmaking algorithms. These algorithms provide the most efficient use of system resources by examining the available resources of the FWAN (a catalog) and comparing them with the resources required by the service to be deployed. In [31], the authors make up an appliance model which treats virtual images to compose environments. After that, they create a virtual solution model by composing virtual appliances and defining its requirements. The obtained virtual solution model is then transformed to a Cloud-specific virtual solution deployment model used to generate a parameterized environment deployment plan.

C. Analysis

The overview of the platforms we presented in subsection II-A allowed us to highlight limitations and constraints regarding heterogeneity of programming languages, communication protocols and/or hosting frameworks support. For supported communication protocols, we focus in our analysis on restrictions that Cloud platforms enforce while a variety of communication protocols and binding implementations are needed for services. For example, *Router* component from Cloud Foundry and *Reverse proxy* component from Heroku allows only HTTP messages. Similarly, in terms of API support, Jelastic supports only Java programming language. *RUN@CLOUD* service of CloudBees platform supports only languages that are based on JVM like Scala, Clojure or Jruby. Furthermore, the specific JVM and APIs offered by GAE are not 100% standard and require adapting applications to deploy to take advantage of the power of the platform. In fact, some Cloud providers compensates the adhesion to a specific programming languages, communication protocols and/or hosting frameworks by continuous development of extensions requested by clients (for proprietary platforms) or proposed by users (for open-source platforms). However, this extension task is quite complex and expensive. It is a development task rather than an integration facility that consists of adding new components without hard coding. To the best of our knowledge, Cloud platforms do not provide all the needed

facilities to support the heterogeneity regarding the usage of programming languages, communication protocols and hosting frameworks for service-based applications. These limitations make difficult the use of these platforms to develop, deploy and host applications that are based on heterogeneous services. Back to *Loan approval* application example, none of these platforms is able to host it as it is described. In addition to that, none of the solutions described in subsection II-B can be adapted in order to do. We believe that Cloud platforms should be architectured to be open and to integrate without hard coding new PaaS resources needed by heterogeneous services that participate in composing heterogeneity of service-based applications.

In this paper, we propose a novel Cloud platform called CloudServ to ensure provisioning of PaaS resources for heterogeneous service-based applications.

III. CloudServ model

A. CloudServ resource model

CloudServ is based on a service delivery model that allows customers to deploy and invoke applications in the Cloud. Such model describes PaaS resources and the relationships between them. A PaaS resource is an elementary platform component able to offer or consume a particular platform service according to its purpose. To model CloudServ resources, we opted to extend the OCCI specification mainly dedicated for IaaS resources modeling and defining a REST API for handling these resources [23]. This specification is already implemented in many managers of Cloud infrastructure. It consists of 3 parts:

- *OCCI core* that defines a meta-model for Cloud [24] (see the top layer of Figure 2),
- *OCCI infrastructure* is an instantiation of the OCCI core meta-model to model infrastructure resources (see middle layer in Figure 2). OCCI infrastructure defines IaaS resources of three types (*i.e.* Network, Compute and Storage) and also classifies the relationship between resources in two types (StorageLink and NetworkLink) [25],
- *OCCI platform* is our proposed extension to the OCCI specification to model PaaS resources. The extension is represented in the bottom layer of Figure 2.

According to our CloudServ model, we have classified PaaS resources into several types:

- *Container* is an engine to host and run services (*e.g.* Apache Axis container),
- *Database* (DB) is storage resource that can be provided by CloudServ (*e.g.* MySQL service),
- *Router* is a resource that provides protocols and messages format transformation and routing (*e.g.* Apache/JK).

We define relations between PaaS resources using different interfaces provided for this purpose:

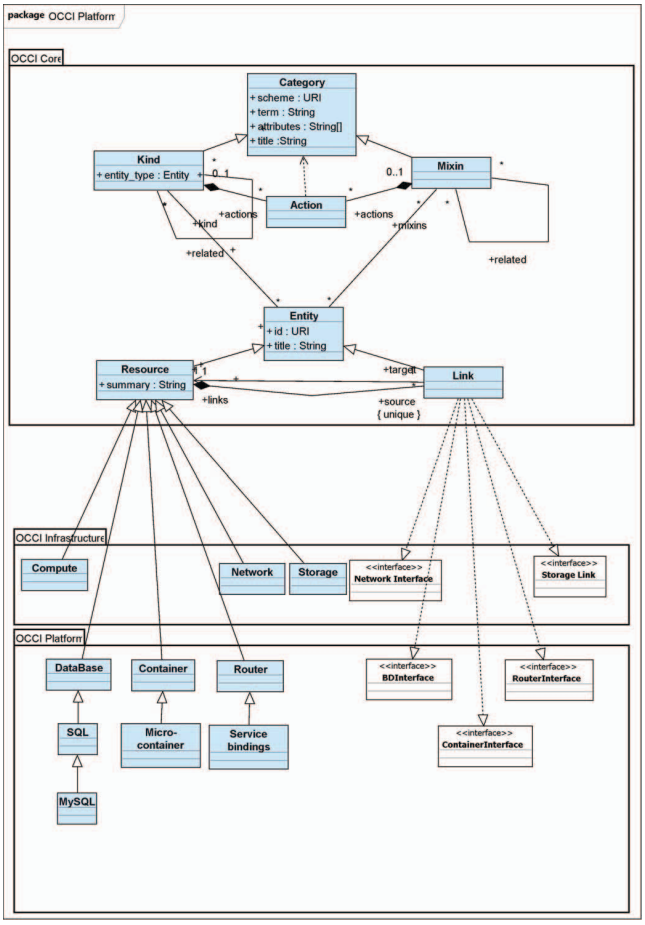

Figure 2. CloudServ model based on OCCI specifications.

- *ContainerInterface* connects a PaaS resource to a *container*,
- *RouterInterface* connects a PaaS resource to a *router*,
- *DBInterface* connects a PaaS resource to a *database*.

PaaS resources with heterogeneous interfaces can be connected using an adequate router.

B. CloudServ provisioning model

CloudServ consists on a set of deployed PaaS resources introduced in subsection III-A. These resources are continuously provisioned along with the arrival of service-based application (un)deployment requests. Concretely, for each service deployment request, one personalized *container* is generated and deployed along with this service. Since we consider several service types (programming languages, bindings, etc.), we are able to generate the correspondent *container* in accordance with the implementation language of the service to deploy and APIs which it depends on. The needed router and other PaaS resources (on-demand resources) are deployed too. So, each time that a service is deployed, additional PaaS resources are also deployed

with. For example, if the developer specifies that its service requires a communication with a new database, a DB node will be deployed with the service *container* in addition to a binding for the communication between them. Similarly, for undeployment requests, provisioned PaaS resources will be also undeployed.

We have designed and implemented multiple deployment processes depending on the nature of the resource to be deployed on CloudServ. Most of CloudServ resources (such as routers) can be deployed on the platform with the launch of a pre-defined script containing all the commands to push these resources in the corresponding Virtual Machine (VM) of the PaaS. Similarly for DB, we have installation and configuration SQL scripts that can be launched if the service to deploy needs a DB. Besides, the task is more complicated for *containers* because they are integrated dynamically in deployment time. In the next section, we detail how *containers* are built adapted to the hosting needs of services and applications to deploy.

IV. CloudServ resources provisioning

A. Provisioning resources for service deployment

Deploying services on classical platforms involves four steps: (1) identifying the needed platform resources, (2) instantiating and initializing them, if needed (3) copying the service resources (code, descriptors, etc.) on platform resources and (4) instantiating and initializing the service. When the service provider requires a specific hosting platform, these steps are adopted in CloudServ. Otherwise, deploying a service on CloudServ consists of (1) identifying the needed platform resources, (2) packaging these resources with the service to be deployed and (3) instantiating and initializing the platform and service resources. This way of doing, consists of dedicating resources for each service to be deployed to host and run its service in a specialized container.

Figure 3 presents our deployment framework. To integrate a micro-container (MC), one must mainly provide for the deployment framework two elements: (1) the service with all its components (code, resources, etc.) and (2) a deployment descriptor that specifies the container options (Figure 3, action 1). The processing module analyzes then the sources, detects the service binding types and associates to the service sources a communication module implementing these bindings and an invocation module to run the service. Bindings are specific types of routers (see Figure 2). The resulting code represents the generated MC code. MCs are specific types of containers (see Figure 2). They are only composed of the necessary modules for the deployed service, no more, no less. We have made several experimentation that shows the efficiency of our MCs compared to classical service containers and Cloud platforms [26] [27]. The framework generates then a MC which hosts the service and implements its bindings dedicated to its communication protocols support as long as they are included in the generic communication package and dedicated the programming language as long as they are included in the generic invocation package (Figure 3, action 2). Adding new communication protocols or programming languages consists of adding the correspondent components in the generic packages (communication and invocation).

It should be noted at this moment of packaging, and based on previous work we have done, that we can also add some optional aspects to the MC from the framework such as mobility [30] and/or monitoring [32]. This will ensure the dynamic mobility of MCs once deployed on CloudServ for eventual optimisations, migrations or other dynamic action. Henceforth we proceed to the deployment task. We deploy the obtained MC in one of the created VMs (Figure 3, action 3). After that, we can invoke it using a client from the SaaS layer (Figure 3, action 4) or even from another deployed MC on CloudServ (Figure 3, action 5). Once deployed, the service is completely autonomous. If it is invoked through its MC, it runs locally and returns the execution result to the requester before processing any other requests. The Cloud infrastructure we used consists of a set of VMs created using OpenNebula from IaaS layer resources as presented in Figure 3.

B. Provisioning resources for application deployment

For service-based applications deployment on CloudServ, we consider two specifications of service composition: (Service Component Architecture) SCA [3] and Business Process Execution Language (BPEL [29]). In this paper, we treat the case of SCA. BPEL treatment can be easily deduced later from our proposal. We are using SCA to define the structure of applications composed of several elementary services, how applications use these services and how they aggregate them together by using an SCA descriptor. Figure 4 express an SCA description fragment of *Loan Approval* application. For each application deployment request, developers have to provide application sources and the correspondant SCA descriptor. The Parser module of the deployment framework slice the composite into several plans. Each plan represents description of one component and the list of other components interacting with it. After that, the plans are sent to the processing manager which treats them as a set of elementary service deployment requests. For each service composing the application, one MC is generated in accordance with service deployment scenario described above (Figure 3, action 2). Adequate routers and bindings are also generated to bind MCs together (using *ContainerInterface* and *RouterInterface*) based on communications protocols supported by the elementary obtained services. Thus, once the MCs are deployed on the platform (Figure 3, action 3), they can participate to

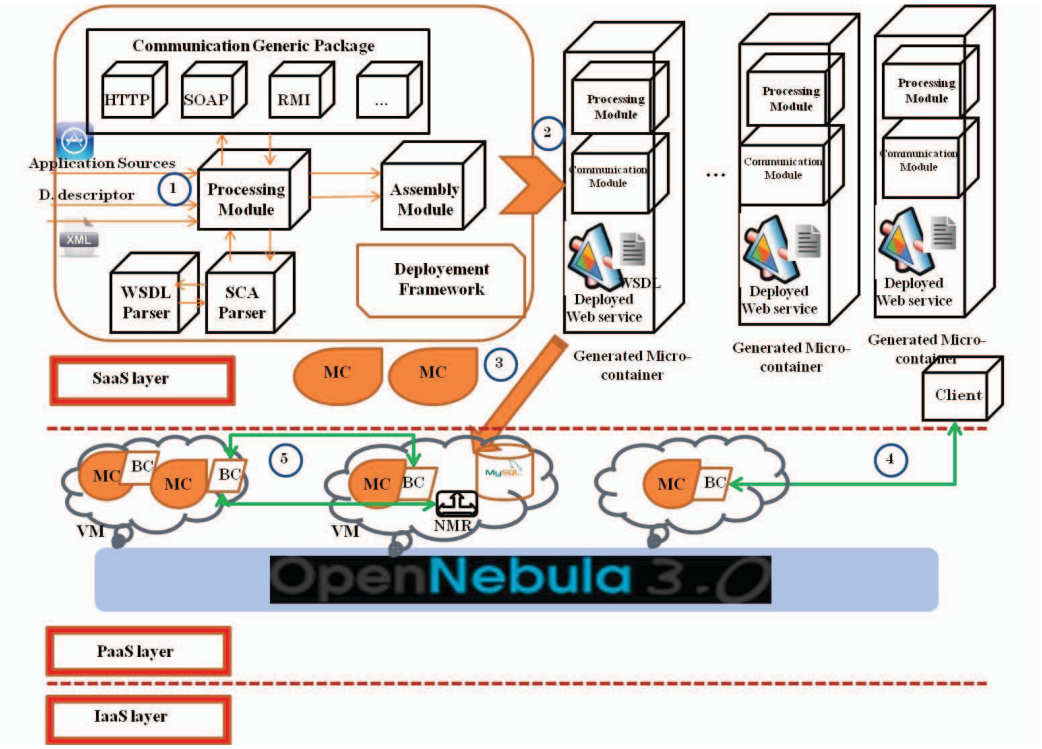

Figure 3. Deployment steps for services and service-based applications.

```
<composite xmlns="http://www.osoa.org/xmlns/sca/1.0"
      targetNamespace="http://example.org" name="...">

      <service name="..."
promote="HandleLoanRequest"/>

      <component name="HandleLoanRequest">
                  <implementation.../>
            <property name="...">...</property>
            <service name="...">...</service>
            <reference name="..."
target="LoanApproavalProcess"/>     </component>

      <component name="LoanApproavalProcess">
                  <implementation.../>
            <property name="...">...</property>
            <service name="...">...</service>
            <reference name="..." .../>
      </component>

      <property name="...">...</property>
      <property name="...">...</property>

      <reference name=""
promote="LoanApproavalProcess"/>
</composite>
```

Figure 4. LoanApproval application SCA descriptor fragment.

a composition scenario to achieve functionality of the applicatio. Specifically, each MC retrieves a set of inputs, runs the service and sends back outputs to the requested clients/services according to interaction semantics implemented in the original service-based application (Figure 3, actions 4 and 5). Obviously, developers can provide only the services that are not already deployed on the platform. They can make a simple reference to the elementary deployed services in the SCA descriptor.

V. Experiments

Besides services heterogeneity support that is achieved by construction, we have experimented CloudServ by comparing it against Cloud Foundry PaaS. We want through these tests demonstrate CloudServ good behaviour for a huge number of deployed services in a Cloud context comparing to a classical Cloud platform to highlight its scalability.
Firstly, we developed a service generator to obtain test collections from a Java service implementing bubble sort algorithm for an array of 20000 doubles. We have deliberately chosen this sort algorithm known for its bad complexity (O (n^2)) and its high memory consumption. We have also developed a client which generates and disseminates deployment requests for these services for Cloud Foundry and CloudServ platforms.

For these experiments, we have considered a couple of criteria that we think essential to evaluate the two PaaS performance:

- *Response time*: Time taken by a service container between request reception instant and response sending

instant,

- *Memory consumption*: Memory size necessary to load and process deployed services in the container after receiving a request.

To perform these tests, we used the Network and Cloud Federation (NCF) experimental platform deployed at TELECOM SudParis France. NCF platform is in constant evolution and has currently: 380 Intel Xeon Nehalem Cores, 1,17 TB RAM and 100 TB as shared storage. We have used OpenNebula resources manager to create our experimental VMs from a personalized template characterized by 2 Cores, 6G RAM, Ubuntu OS and Java Runtime Environment (JRE). We varied the experiments under several scenarios defined in advance to identify the behavior of both platforms in many situations as possible. Indeed, we conducted experiments by deploying elementary services on both PaaS before repeating the same scenarios for service-based applications. Results of these two scenarios are detailed below.

A. Deployed service experiments

For elementary service tests, we distinguished between two scenarios. We first installed the two PaaS on a single VM. Then, we redo the same tests with a multi-VM installation.

1) Single VM installation: In this scenario, we used two VMs. We installed the two respective PaaS on these VMs before starting to deploy same services on both sides and take mesurements. Figure 5 shows response time curve of the two PaaS regarding the evolution of the number of deployed services. In this experiment, we calculate the average of response time for invocation of 10% of deployed services.

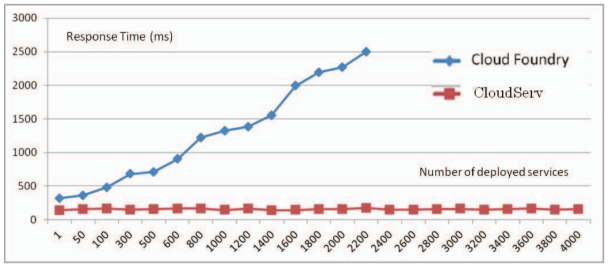

Figure 5. Average of response time of the invocation of 10% of deployed services (1VM).

We note that for the obtained curve, CloudServ response time is constant while Cloud Foundry response time increases regarding the number of deployed services. Effectively, a huge number of deployed services affects performance of Cloud Foundry internal service container (Tomcat server for Java services). In contrast, CloudServ performance is virtually unchanged and look like MCs experiments detailed in [26]. This amounts to the time required for an invocation of a service that is deployed on a dedicated MC added to negligible time for routing on PaaS. To validate these interpretations, we redo this experiment by invoking only one random service among all deployed services and we found the same curve shape. Furthermore, it should be noted that for this scenario, we could not deploy more than 2400 services on Cloud Foundry because, reached this stage, it has consumed all the memory resources of its VM as opposite to CloudServ VM where we have deployed more than 4000 services (see Figure 6). Naturally, this threshold increases

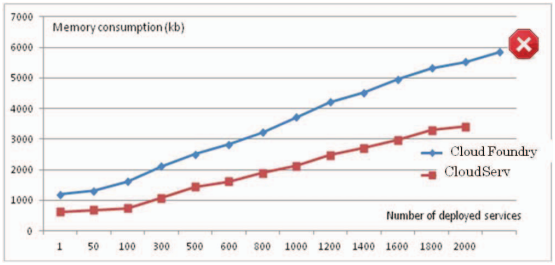

Figure 6. Memory consumption curve (1VM).

by increasing VM template but Cloud Foundry still crashes when approaching to reach VM limit.

2) Multi-VM installation: This scenario consists on the installation and configuration of the two PaaS on several VMs. For this scenario, we have repeated the same tests performed previously on one VM. Figure 7 shows the average of response time values for invocation of 10% of deployed services.

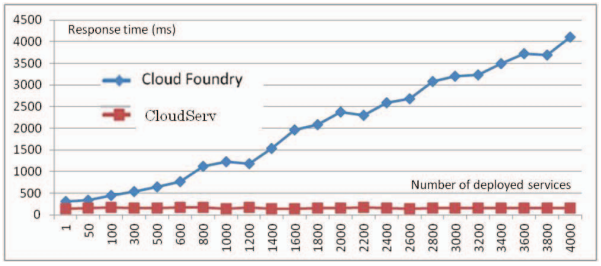

Figure 7. Average of response time of the invocation of 10% of deployed services (Multi-VMs).

The curve has the same shape with that of the previous scenario with a slight increase in response time. This increase is also seen in one random service invocation measurements too. This is due to the overhead caused by the interaction cost between the various platform components that are now distributed. Thus, communication between them became more costly. We also notice that difference between the two PaaS performance is more important than in the first scenario because the overhead of routing between *Cloud Controller* and *DEA* components at the moment at they are distributed on. Finally, we note that we have reached the limit observed in the last scenario of the number of deployed services on Cloud Foundry. Even for a huge number of deployed services, CloudServ still has better performance even for memory consumption resources illustrated in Figure 8.

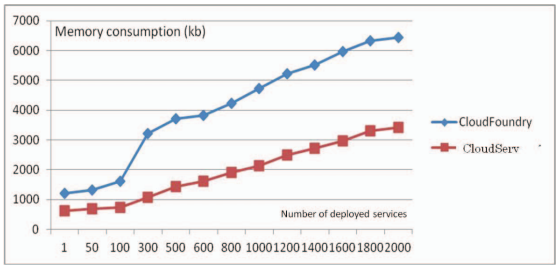

Figure 8. Memory consumption curve (Multi-VMs).

B. Deployed application experiments

To perform these tests, we change our elementary sort service model in a service-based application. We modified our service generator to generate two types of test collections called TC1 and TC2. Services from TC1 generates an array of 20000 doubles, instantiates it with random values and send it to services from TC2. Services from TC2 applies bubble sort algorithm to sort the array and returns it to services from TC1 to display. Deployment of this application consists on the deployment of a couple of services: one from TC1 and another from TC2. Since we do not have large differences between the two scenarios of the first part of the tests, we performed these experiments directly with a multi-VM configuration. Figure 9 shows response time evolution for the two PaaS corresponding to an application invocation.

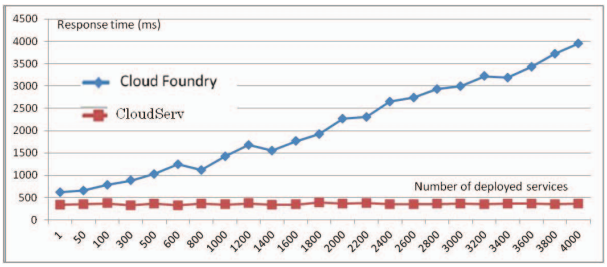

Figure 9. Response time-10% of applications average.

Curve's shape is equivalent with those of the first part of experiments. However, the difference between the performances of two PaaS is more effective especially for huge number of deployed services. Indeed, when this number grows up, Cloud Foundry duplicates its containers through *DEA* duplication in accordance to its horizontal duplication algorithm to scale [8] [9]; and it is not uncommon for the two application services to be deployed on different containers or worse on different VMs. To deal with this, Cloud Foundry uses registers for indexing services and routing messages between these services managed on *Cloud controllers* components which affect response time. Moreover, the inter-container communication provides additional cost because the containers deals with context management for these interactions. Opposed to this, interactions between micro-containers, is assimilated to remote calls by specifying VM IP where the service is hosted as well as the listening port of the service in the MC. These performance differences are more distinct in the response time curve for one random application illustrated in Figure 10.

Memory consumption values of two PaaS are illustrated in

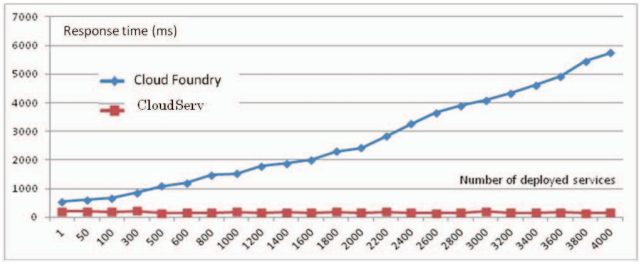

Figure 10. Response time-1 random application.

Figure 11. The noticed indexation, routage and communication overhead affects memory resources consumption.

It is clear that Cloud Foundry requires significantly more memory resources as CloudServ to host and run the same number of applications.

VI. Conclusion

In this paper, we highlighted portability and compatibility constraints imposed by classical Cloud providers and platforms making development and deployment tasks difficult and delicate for users. We have proposed a new prototype of platform as-a-service (PaaS) called CloudServ that resolves these drawbacks. CloudServ is based on reuse of the scalable service micro-containers introduced as a part of our previous work. Only necessary resources to implement service binding types, such as communication protocols, are selected from the deployment framework of the system and encapsulated in the generated micro-container to host the deployed service. We also presented the CloudServ model and described the different discussed use case for CloudServ resources construction and deployment. To validate our proposal, we have evaluated and compared performance and behavior of our platform versus Cloud Foundry. These experiments are conclusive and demonstrate

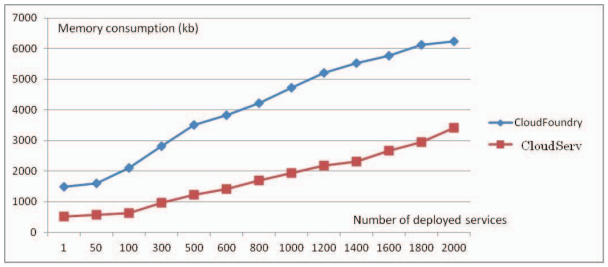

Figure 11. Memory consumption curve-applications deployment.

the scalability and the good behavior of CloudServ versus Cloud Foundry given their respective response time and memory consumption records. In the near future, we plan to include new features to CloudServ to support deployment of service-based application described using WS-BPEL.

REFERENCES

[1] G. Cliquet. *Method of innovation in the era of Web 2.0*. PhD thesis, Paris Institute of Technologie, Paris, France, 2011.

[2] I. Foster, Y. Zhaoand, I. Raicuand, and S. Lu. *Cloud computing and grid computing 360-degree compared*. In The IEEE Grid Computing Environments (GCE'08), Austin, USA, 2008.

[3] J. Marino and M. Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 1st edition, 2009.

[4] Apex: Salesforce on-demand programming language and framework, http://developer.force.com/ (2012).

[5] D. Sanderson. *Programming Google App Engine: Build and Run Scalable Web Apps on Google's Infrastructure*. O'Reilly Media, Inc., 1st edition, 2009.

[6] A. Skonnard and K. Brown. *An Introduction to Windows Azure platform AppFabric for Developers*. November 2009.

[7] J. Dean and S. Ghemawat. *Mapreduce: Simplified data processing on large clusters*. In Sixth Symposium on Operating System Design and Implementation (OSDI'04), San Francisco, USA, 2004.

[8] Cloud Foundry. Cloud foundry. ttp://www.cloudfoundry.com/ (2012).

[9] Cloud Foundry. Cloud foundry. ttp://www.cloudfoundry.org/ (2012).

[10] Cloud Foundry. Cloud foundry on github. https://github.com/cloudfoundry (2012).

[11] CloudBees. Cloudbees. http://www.cloudbees.com/ (2012).

[12] Cloud Comments. Clouds comments.net. http://cloudcomments.net/author/grapesfrog/ (2012).

[13] Heroku. Heroku platform. http://www.heroku.com/ (2011).

[14] 15. Jenkins-CI. Jenkins-ci. http://jenkins-ci.org/ (2011).

[15] OpenShift. Red hat openshift. https://openshift.redhat.com/app/ (2011).

[16] Xebia. Xebialabs deployement lifecycle management. http://blog.xebia.fr/2011/04/13/lancement-du-projet-platform-as-a-service- cloud-foundry-de-spring-source/ (2011).

[17] Amazon. Amazon web services. http://aws.amazon.com/fr/ (2011).

[18] C.Chrysoulas, E.Haleplidis, R.Haas, S.Denazis, O.Koufopavlou. *Applying a Web-Service-Based model to Dynamic Service-Deployment*. In International Conference on Intelligent Agents, Web Technology and Internet Commerce (IAWTIC'05), Vienna, Austria, 2005.

[19] P.Watson and C.Fowler. *An Architecture for the Dynamic Deployment of Web Services on a Grid or the Internet*. School of Computing Science, University of Newcastle upon Tyne. Technical Report Series CS-TR-890, 2005.

[20] J.Kirschnick, M.Alcaraz Calero, L.Wilcock and N.Edwards. *Toward an Architecture for the Automated Provisioning of Cloud Services*. In IEEE communications Magazine, Topics in Network and Service Management, December 2010.

[21] D. Frost. *Using Capistrano*. In Linux Journal, vol.177 p8, 2009.

[22] D.Solutions, ControlTier. http://controltier.org/wiki/MainPage (2012).

[23] Open Cloud Computing Interface - OCCI. Online: http://occi-wg.org/ (2012).

[24] T. Metsch, A. Edmons, and R. Nyren. *Open cloud computing interface - core*. http:www.gridforum.org/Public Comment Docs/Documents/2010- 12/ogf draft occi core.pdf, 2010.

[25] Open Cloud Computing Interface-OCCI. Online: http://www.ogf.org/documents/GFD.184.pdf (2012).

[26] S. Yangui, M. Mohamed, S. Tata, and S. Moalla. *Scalable service containers*. In IEEE International Conference on Cloud Computing Technology and Science (IEEE CloudCom'11), Athenes, Greece, 2011.

[27] M. Mohamed, S. Yangui, S. Moalla, and S. Tata. *Service micro-container for service-based applications in cloud environments*. In IEEE WETICE, Paris, France, 2011.

[28] R.L. Grossman. *The case for cloud computing*. IT Professional, 11 Issue:2: pp 2327, 2009.

[29] B.Juric. *Business Process Execution Language for Web Services BPEL and BPEL4WS*. 2nd Edition. ISBN:1904811817, 2006.

[30] A. Omezine, S.Yangui, N.Bellamine and S.Tata. *Mobile Service micro-container for Cloud environments* In IEEE WETICE, Toulouse, France, June 25-28, 2012.

[31] A.Konstantinou, T.Eilam, M.Kalantar, A.Totok and W.Arnold. *An architecture for virtual solution composition and deployment in infrastructure clouds*. In 3rd international workshop on Virtualization technologies in distributed computing (VTDC'09), NY, USA, 2009.

[32] M.Mohamed, D.Belad and S.Tata. *How to Provide Monitoring Facilities to Services When They Are Deployed in the Cloud?*. In International Conference on Cloud Computing and Services Science (CLOSER'12), Porto, Portugal, 2012.