Service-based applications provisioning in the cloud

Sami Yangui

DOCTORAT EN CO-ACCREDITATION
TELECOM SUDPARIS ET L'UNIVERSITE EVRY VAL D'ESSONNE

Spécialité: Informatique

Ecole doctorale: Sciences et Ingénierie

Présentée par

Sami Yangui

Pour obtenir le grade de
DOCTEUR DE TELECOM SUDPARIS

SERVICE-BASED APPLICATIONS PROVISIONING IN THE CLOUD

Soutenue le 02/10/2014

devant le jury composé de :

Directeur de thèse :

M. Samir Tata	Professeur, Institut Mines-Télécom, Télécom SudParis, France.

Rapporteurs :

Mrs. Daniela Grigori	Professeur, Université Paris-Dauphine, France.
M. Olivier Perrin	Professeur, Université de Lorraine, France.

Examinateurs :

M. Bruno Defude	Professeur, Institut Mines-Télécom, Télécom SudParis, France. (Président du jury)
M. Jean-Pierre Lorré	Directeur R&D, Linagora Toulouse, France.
M. Ernesto Exposito	Maître de conférences, INSA Toulouse, France.

Thèse numéro : 2014TELE0024

"Citius, Altius, Fortius"
Devise Olympique...

Dédicaces

A l'âme des grands parents,

A toi *Habiba*, partie si jeune, très tôt, trop tôt. Chaque jour, tu me hantes, tu me manques. C'est à toi que je pense en premier, c'est à toi que je dois tout, absolument tout.

A toi *Mohamed*. Ton coeur d'or a cessé de battre, voilà déjà 9 mois. La douleur de ton départ est marquée dans ma mémoire, dans ma chair, je ne pourrai jamais l'effacer, d'ailleurs je n'aurai pas envie. Tu avais si hâte que je terminais ce périple. Tu m'avais fais la promesse d'assister au jour de ma consécration si ta santé te le permettait, c'était sans compter sur la volonté de Dieu qui t'a rappelé si vite.

A toi aussi *Mohamed*. Le grand-père paternel que je n'ai jamais connu.

Reposez en paix.

A la grand-mère paternelle,

Si douce, si paisible et si pieuse.

Aux parents,

Pour ce que vous m'avez inculqué, les sacrifices que vous avez consentis pour moi, mais encore pour le soutien, la confiance et la présence.

A koukia,

Ma chère et tendre, mon compagnon de route et la lumière de mon chemin.

A Yassine,

Ma plus belle réussite dans la vie.

A toute la famille et aux amis.

Remerciements

La reconnaissance est la mémoire du coeur.
Hans Christian Andersen, écrivain danois (1805-1875)

Je remercie tout d'abord les membres du jury. Un très grand merci à Bruno Defude, Professeur et Directeur adjoint de la Recherche et des formations doctorales à Télécom SudParis, de m'avoir accordé l'honneur d'être le président de mon jury.

Je remercie aussi Daniela Grigori, Professeur à l'Université Paris-Dauphine, et Olivier Perrin, Professeur à l'Université de Lorraine d'avoir accepté la fastidieuse tâche de rapporter ce travail.

Je remercie également Ernesto Exposito, Maître de conférences à l'INSA Toulouse et Jean-Pierre Lorré, Directeur R&D à Linagora d'avoir accepté d'examiner mon travail.

Et puis, je remercie surtout celui auprès de qui j'ai découvert la recherche : Samir Tata, mon superviseur, qui a eu confiance en mes capacités et a tout fait durant ces années pour m'assurer un environnement propice au travail. Samir, ta confiance, ton soutien, ta disponibilité, tes encouragements et ton dynamisme au quotidien m'ont permis d'être là où je suis aujourd'hui. Je te suis reconnaissant. Sache que ta perception, ton œil critique, ta minutie et ta rigueur m'ont marqué à jamais. C'est mon exemple à suivre.

Finalement, je remercie tous les membres de l'équipe qu'ils soient anciens ou nouveaux, doctorants ou permanents. L'aventure avec eux était une riche expérience, j'en garderai de très bons souvenirs. Je remercie plus particulèrement Brigitte Houassine, notre chargée de gestion, pour son accompagnement et sa bienveillance.

Résumé

Le Cloud Computing est de plus en plus utilisé pour le déploiement et l'exécution des applications en général et des applications à base de services en particulier. Les applications à base de services sont décrites à l'aide du standard Service Component Architecture (SOA) et consistent à inter-lier un ensemble de services élémentaires et hétérogènes en utilisant des spécifications de composition de services appropriées telles que Service Component Archi tecture (SCA) ou encore Business Process Execution Language (BPEL). Provisionner une application dans le Cloud consiste à : (1) allouer les ressources dont elle a besoin pour s'exécuter et (2) déployer ses sources sur les ressources allouées. Cependant, les solutions Cloud existantes sont limitées en termes de plateformes d'exécution. Ils ne peuvent pas toujours satisfaire la forte hétérogénéité des composants des applications à base de services. Pour remédier à ces problèmes, les mécanismes de provisioning des applications dans le Cloud doivent être reconsidérés. Ces mécanismes doivent être assez flexibles pour supporter la forte hétérogénéité des composants sans imposer de modifications et/ou d'adaptations du côté du fournisseur Cloud. Elles doivent également perme ttre le déploiement automatique des composants dans le Cloud. Si l'application à déployer est mono-composant, le déploiement est fait automatiquement et de la même manière, et ce quelque soit le fournisseur Cloud choisi. Si l'application est à base de services hétérogènes, des fonctionnalités appropriées doivent être mises à la disposition des développeurs pour qu'ils puissent définir et créer les ressources nécessaires aux composants avant de déployer l'application. Dans ce travail, nous proposons une approche appelée SPD permettant le provisioning des applications à base de services dans le Cloud. L'approche SPD est constituée de 3 étapes : (1) découper des applications à base de services en un ensemble de services élémentaires et autonomes, (2) encapsuler les services dans des micro-conteneurs spécifiques et (3) déployer les micro-conteneurs dans le Cloud. Pour le découpage, nous avons élaboré un ensemble d'algorithmes formels assurant la préservation de la sémantique des applications une fois découpées. Pour l'encapsulation, nous avons réalisé des prototypes de conteneurs de services permettant l'hébergement et l'exécution des services avec seulement le minimum des fonctionnalités nécessaires. Pour le déploiement, deux cas sont traités i.e. déploiement sur une infrastructure Cloud (IaaS) et déploiement sur une plateforme Cloud (PaaS). Pour automatiser le processus de déploiement, nous avons défini : (i) un modèle de description des ressources unifié basé sur le standard Open Cloud Computing Interface (OCCI) permettant de décrire l'application et ses ressources d'une manière générique quelque soit la plateforme de déploiement cible et (ii) une API appelée COAPS implémentant ce modèle et permettant de l'approvisionnement et la gestion des applications en utilisant des opérations génériques quelque soit la plateforme cible.

Mots clés: Application à base de service - Approvisionnement de ressource Cloud - Cloud Computing, Micro-conteneur de services - Modélisation de ressource Cloud

Abstract

Cloud Computing is a new supplement, consumption, and delivery model for IT services based on Internet protocols. It is increasingly used for hosting and executing applications in general and service-based applications in particular. Service-based applications are described according to Service Oriented Architecture (SOA) and consist of assembling a set of elementary and heterogeneous services using appropriate service composition specifications like Service Component Architecture (SCA) or Business Process Execution Language (BPEL).

Provision an application in the Cloud consists of allocates its required resources from a Cloud provider, upload source codes over their resources before starting the application. However, existing Cloud solutions are limited to static programming frameworks and runtimes. They cannot always meet with the application requirements especially when their components are heterogeneous as service-based applications. To address these issues, application provisioning mechanisms in the Cloud must be reconsidered. The deployment mechanisms must be flexible enough to support the strong application components heterogeneity and requires no modification and/or adaptation on the Cloud provider side. They also should support automatic provisioning procedures. If the application to deploy is mono-block (e.g. one-tier applications), the provisioning is performed automatically and in a unified way whatever is the target Cloud provider through generic operations. If the application is service-based, appropriate features must be provided to developers in order to create themselves dynamically the required resources before the deployment in the target provider using generic operations.

In this work, we propose an approach (called SPD) to provision service-based applications in the Cloud. The SPD approach consists of 3 steps: (1) Slicing the service-based application into a set of elementary and autonomous services, (2) Packaging the services in micro-containers and (3) Deploying the micro-containers in the Cloud. Slicing the applications is carried out by formal algorithms that we have defined. For the slicing, proofs of preservation of application semantics are established. For the packaging, we performed prototype of service containers which provide the minimal functionalities to manage hosted services life cycle. For the deployment, both cases are treated i.e. deployment in Cloud infrastructure (IaaS) and deployment in Cloud platforms (PaaS). To automate the deployment, we defined: (i) a unified description model based on the Open Cloud Computing Interface (OCCI) standard that allows the representation of applications and its required resources independently of the targeted PaaS and (ii) a generic PaaS application provisioning and management API (called COAPS API) that implements this model.

Keywords: Cloud Computing - Cloud resource modeling - Cloud resource provisioning - Service-based application - Service micro-container

List of Publications

Peer-Reviewed Journal Articles

[I] Sami Yangui and Samir Tata. An OCCI Compliant Model for PaaS Resources Description and Provisioning. The Computer Journal. Oxford Journals: Science & Mathematics, United Kingdom. Accepted 2014, In press.

[II] Sami Yangui and Samir Tata. The SPD approach to deploy service-based applications in Cloud platforms. Concurrency and Computation: Practice and Experience. John Wiley & Sons Ltd, United Kingdom. Accepted 2014, In press.

[III] Sami Yangui, Iain-James Marshall, Jean-Pierre Laisne and Samir Tata. CompatibleOne: The Open Source Cloud Broker. Journal of Grid Computing. Springer Journal. vol.12, Issue 1, pp. 93-109, Springer Netherlands, 2014.

Peer-Reviewed Conference Articles

[IV] Sami Yangui, Kais Klai and Samir Tata. Deployment of Service-based Processes in the Cloud using Petri Net Decomposition. The 22nd International Conference on COOPERATIVE INFORMATION SYSTEMS (CoopIS 2014), CoopIS'14, Amantea, Italy, October 2014.

[V] Sami Yangui, Marwa Ben Nasrallah and Samir Tata. PaaS-independent approach to provision appropriate Cloud resources for SCA-based applications deployment. The 9th International Conference on Semantics, Knowledge & Grids, SKG'13, Beijing, China, October 2013.

[VI] Mohamed Sellami, Sami Yangui, Mohamed Mohamed and Samir Tata. PaaS-independent Provisioning and Management of Applications in the Cloud. IEEE International Conference on Cloud Computing, CLOUD'13, Santa Clara Marriott, CA, USA, June-July 2013.

[VII] Sami Yangui, Samir Tata. CloudServ: PaaS resources provisioning for service-based applications. IEEE International Conference on Advanced Information Networking and Applications, AINA'13. Barcelona, Spain, March 2013.

[VIII] Aya Omezzine, Sami Yangui, Narjes Bellamine and Samir Tata. Mobile Service Micro-containers for Cloud Environments. IEEE International Conference on Collaboration Technologies and Infrastructures, Wetice'12. Toulouse, France, June 2012.

[IX] Sami Yangui, Samir Tata. Paas Elements for Hosting Service-based Applications. International Conference on Cloud Computing and Services Science, CLOSER'12. Porto, Portugal, April 2012.

[X] Sami Yangui, Mohamed Mohamed, Samir Tata and Samir Moalla. Scalable service containers. IEEE International Conference on Cloud Computing Technology and Science, CloudCom'11, Athenes, Greece, November-December 2011.

[XI] Mohamed Mohamed, Sami Yangui, Samir Moalla and Samir Tata. Service micro-container for service-based applications in Cloud environments. IEEE International Conference on Collaboration Technologies and Infrastructures, Wetice'11. Paris, France, June 2011.

Table of contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Context

Cloud Computing is a model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g. networks, servers, storage resources, applications, services, etc.). These resources should be swiftly provisioned and released with minimal management effort and service providers interactions [11].

In this work, we are interested in service-based applications provisioning in Cloud environments. These applications are built according Service Oriented Architecture (SOA) which is a software architecture and applications design principle that starts with an interface definition and builds the entire application topology as a topology of interfaces, interface implementations and interface calls [12].

Service-based applications consist in assembling a set of elementary services using appropriate service composition specifications such as Service Component Architecture (SCA) [13], Business Process Model and Notation (BPMN) [14] or Web Services Business Process Execution Language (BPEL) [15]. Service-based applications are built from components and services that maybe heterogeneous in the sense that they (1) are not all implemented using the same programming languages (e.g. C++, Java, etc.), (2) do not support all the same communication protocols (e.g. RMI, SOAP/HTTP, etc.) and/or (3) do not run on the same hosting frameworks (e.g. POJO VM, .NET framework, etc.).

Provisioning a service-based application in the Cloud consists of: (1) allocation of adequate resources to host and execute the application and (2) upload of the application artifacts (e.g. binary code) on the allocated resources. This provisioning requires then the delivery of appropriate frameworks and specific runtimes supporting the heterogeneities presented above. However, existing Cloud platforms are limited to specific programming frameworks and runtimes. For example, Jelastic PaaS do not support .NET framework provisioning and cannot therefore host and execute .NET-based applications. Furthermore, each Cloud platform describe, manage and provision these resources in a proprietary way. These differences can be explained by several

reasons such as the huge variety of manipulated resources and applications and the absence of universal standards for both applications and Cloud resources. For example, to deploy a Java Web application in Cloud Foundry, the developer have only to provide the application byte code. The allocation of the hosting tomcat server is performed implicitly by the PaaS based on the application type. While, deploying the same application in Jelastic requires: (1) the creation (manually) of the hosting environment containing a tomcat instance by the developer, (2) the upload of the application and (3) the linking of the application to the created environment for the concrete deployment.

1.2 Thesis research issues and motivations

On one side, SOA developers manipulate various and heterogeneous application components which: (1) do not use necessarily the same programming languages, (2) do not support all the same communication protocols and/or (3) do not run on the same hosting frameworks. On the other side, taxonomy of Cloud Computing shows that all the existing systems are limited to specific hosting environments, programming frameworks and runtimes. Each Cloud system provides a finite and limited set of hosting and execution resources (e.g. execution engines, message routers, etc.) and services (e.g. storage services, logging services, etc.). For example, deploying BPEL processes (respectively SCA-based applications) in the Cloud requires provisioning, among others, specific execution engines such as Apache ODE (respectively Apache Tuscany). These limitations impose constraints and makes the use of those systems difficult, since SOA developers need to use the related programming languages and execution frameworks before using the Cloud [16]. Capabilities of Cloud platforms are limited in terms of hosting and execution resources and cannot meet all the time the high heterogeneity of application components.

Some of the existing solutions (e.g. Cloud4SOA, mOSAIC, etc.) propose to provide dedicated frameworks for SOA applications, or to allow developers to install and configure themselves required execution frameworks when deploying the application components (e.g. [17] for Cloud Foundry PaaS), these solutions delegate installation and configuration tasks to developers which complicates significantly the deployment, constitutes a step backward and is inconsistent with the Cloud business model. Other solutions (e.g. Cloud Foundry, WSO2 Stratos, etc.) compensate this adhesion to specific programming languages, communication protocols and/or hosting frameworks by continuous development of extensions requested by end users (for proprietary solutions) or proposed by developers (for open-source solutions). Nevertheless, 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.

In addition to these constraints, we noticed another type of restrictions when deploying service-based applications in the Cloud. Such applications are often distributed, so it is not uncommon to deploy their components separately on multiple

Cloud platforms for example. However, application required resources are provisioned by Cloud platforms in a specific way. Each Cloud system has proprietary description models to describe, manage and provision applications and their hosting resources. This is also reflected in the heterogeneity of the user APIs implementing these description models (e.g. proprietary operations, specific provisioning scenarios, etc.). These limitations are forcing developers to adapt their applications and provisioning procedure when they provision them into several Cloud platforms and/or when they move from a Cloud platform to another. These limitations cause application portability issues which impedes operating such applications in Cloud context. Cloud applications portability is a concept that refers to the ability to move applications between Cloud vendors with a minimum level of integration issues. Cloud applications portability enables the re-use of application components across Cloud platforms and services [18]. Application portability limitations lead to restriction problems, compatibility drawbacks and vendor lock-in that make operating service-based applications difficult in the Cloud due to the differences on used resources description models and user APIs.

To address these issues, we define, in this thesis, a novel approach to provision service-based applications in Cloud environments. To do this, we propose to design and implement appropriate mechanisms to support the high heterogeneity of the applications components and generic operations independent from the target Cloud environment for applications deployment.

1.3 Thesis objectives and principles

In this thesis, we aim at defining an approach to provision automatically service-based applications in Cloud environments. This approach aims at addressing highlighted SOA applications heterogeneity limitations and portability issues in the Cloud. Our approach covers applications described according SOA, and particularly business processes modeled through BPEL or BPMN processes and applications whose services compositions can be modeled as directed graphs such as SCA-based applications.

Our approach consists in defining and implementing new provisioning mechanisms that are **flexible enough to support the deployment of high heterogeneous service-based application components**. These mechanisms must also allow **application portability** to enable automatic and unified provisioning and management procedures whatever is the target Cloud without any modifications and/or adaptations on the Cloud environment side.

To support application components heterogeneity, we propose to slice the applications into a set of elementary and autonomous services before allocating a dedicated and appropriate environment for each one of the obtained services. Applications slicing in elementary services seeks to facilitate deployment task and heterogeneity constraints satisfaction when instantiating hosting Cloud environments. Indeed, it is difficult to meet Cloud provided resources with such varied application components

hence our idea of slicing. After that, if the provisioning of required Cloud resources for hosting and execution of these services is not supported by the target Cloud environment, we propose to perform dedicated service containers wherein we package sliced services and required execution resources.

To enhance Cloud portability, we define a unified application and resources description model and a generic API implementing this model. Based on our introduced model and its correspondent API, we are able to describe, provision and manage applications and allocated resources on the same way whatever is the target Cloud infrastructure or platform.

Finally, it should be noted that based on these principles, the execution of the deployed service-based applications are assimilated to services choreographies instead of the initial services orchestrations which is more appropriated for decentralized environments such as the Cloud.

1.4 Thesis Contributions

In order to achieve our stated objectives, we define a novel approach that we called SPD to perform the provisioning procedure. The SPD approach requires no modification and/or adaptation on Cloud provider side and enable applications portability thanks to the generic resources description model that we define for both application and Cloud resources.

The SPD approach consists of three steps:

1. Slicing the application into a set of elementary services,
2. Packaging the resulted services into service micro-containers,
3. Deploying the micro-containers in a target Cloud environment.

If the application to deploy is mono-block (e.g. Java Web application), we perform directly the third step of the approach. Else, if the application has several components, we perform the first step of the approach to slice it before. Note that the second step is performed only when the target Cloud platform do not support provisioning required Cloud resources for hosting and execution of sliced services. The three SPD approach steps are detailed in the following.

1.4.1 Step 1: Slice the service-based application

The slicing step is based on the application deployables and descriptor. Application deployables are all necessary artifacts (e.g. ZIP file, EAR file, configuration scripts, etc.) needed to run the application while the application descriptor, often an XML-based document, is a sort of contract describing how to invoke the application and giving details regarding its several components, bindings and interactions between them. The principle of this step is to cut and divide the service-based application into

an equivalent set of autonomous and operational elementary services while ensuring the preservation of their initial business semantics. The execution of these resulting services ensures the same functionality of the initial application.

1.4.2 Step 2: Package the services to deploy

In this step, we package each one of the obtained services from the slicing step in a particular type of service container. Only one service and necessary resources to implement its binding types such as communication protocols and its required facilities such as migration, elasticity or monitoring are packaged in a dedicated container that we called service micro-container. The micro-containers are generated from the packaging framework based on the provided service.

1.4.3 Step 3: Deploy the packaged services

Once the services are packaged in micro-containers, we can deploy them in a target Cloud environment. Depending on the choice of the developer, the micro-containers can be deployed on Cloud infrastructure (IaaS) or Cloud platform (PaaS). For an IaaS deployment, service micro-containers are placed on virtual machines as standalone applications. For a PaaS deployment, we defined a PaaS-independent platform and application resources description model based on the Open Cloud Computing Interface (OCCI) to enable applications portability. We also performed a REST API called COAPS implementing this model and enabling unified and automatic provisioning through generic operations.

1.5 Thesis outline

This thesis includes 7 chapters:

In Chapter 2, we introduce a set of definitions and basic concepts before presenting the work related to our thesis. We study results of different collaborative research projects (e.g. Cloud4SOA, mOSAIC, etc.), tentatives of standardization (e.g. TOSCA, CAMP, etc.) and existing solutions for service-based applications description and provisioning in the Cloud. This analysis allows us to highlight the existing limitations and justify the need of novel appropriate mechanisms for service-based applications provisioning in the Cloud.

Chapters 3, 4 and 5 are the core of our thesis, which elaborate our defined SPD approach to provision service-based applications in Cloud environments. Each one of these Chapters details a step of our SPD approach. Concrete illustrative examples are provided at the end of each Chapter.

In Chapter 3, we present our performed work to achieve the first step of the SPD approach. In this Chapter, we define and comment a set of formal algorithms that slices service-based applications based on their type (i.e. business processes or

applications based on service compositions). The defined slicing algorithms allow the preservation of the applications business semantics.

In Chapter 4, we present the architecture of the packaging framework and its execution process to package sliced services in appropriate service micro-containers. Extended variants of packaging framework supporting the packaging of non-functional properties such as migration or monitoring in the micro-containers are also described.

In Chapter 5, we present our proposed applications and platform resources description model. We also describe the COAPS API specifications which implements this model. Our defined model allows applications and/or services deployment through seamless interactions with different and heterogeneous PaaS and address applications portabiliy issues.

In Chapter 6, we present implementations details and used technologies to realize each step of the SPD approach. We discuss also the experimentations results that we have conducted to evaluate our service micro-container performances against classical service containers in Cloud environments. In the last part of the Chapter, we detailed two realistic use cases of applications provisioning in Cloud platforms based on our findings.

Finally, in Chapter 7, we summarize our work and give an outlook of the future work.

Chapter 2

State of the Art

Contents

This Chapter is organized as follows: We introduce a set of definitions and basic concepts related to our work in Section 2.1. Then, we present the criteria that we have selected to evaluate related works in Section 2.2. After that, we discuss results of collaborative research projects and existing approaches for applications provisioning in the Cloud in Section 2.3. Finally, we study and compare existing approaches for Cloud resources description in Section 2.4.

2.1 Background

In this Section, we introduce definitions and basic concepts related to our work.

2.1.1 Cloud Computing

The American National Institute of Standards and Technology (NIST) defined Cloud Computing as a new emerging model for enabling ubiquitous, convenient, on-demand network access to shared pool of configurable computing resources (e.g. networks, servers, storage resources, applications, services, etc.). These resources should be swiftly provisioned and released with minimal management effort [11].

Based on [19], Cloud Computing is defined as 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. The associated delivery models to Cloud Computing are: Infrastructure as-a-Service (IaaS), Platform as-a-Service (PaaS) and Software as-a-Service (SaaS).

IaaS provides services that furnish to the consumer processing, storage, networks, and other fundamental computing resources where the consumer is able to deploy and run arbitrary software, which can include operating systems and applications. At IaaS level, the consumer does not manage or control the underlying Cloud infrastructure but has control over operating systems, storage, and deployed applications [11]. Examples of IaaS are Amazon AWS[1], Rackspace Open Cloud[2], Google Compute Engine[3], etc.

PaaS provides services that furnish to the consumer appropriate resources to deploy in the Cloud infrastructure consumer-created or acquired applications implemented using programming languages, libraries, services and tools supported by the provider [11]. PaaS consists of a re-usable framework, which provides one or more application platform components as a service. Examples of PaaS are Cloud Foundry[4], OpenShift[5], Windows Azure[6], Google App Engine[7], Heroku[8], Jelastic[9], etc.

SaaS provides services that furnish distributed software over the Internet. SaaS supports a model for access and execution of software deployed over PaaS systems [11].

[1] aws.amazon.com
[2] rackspace.com/cloud
[3] cloud.google.com/compute
[4] cloudfoundry.org
[5] openshift.com
[6] azure.microsoft.com
[7] appengine.google.com
[8] heroku.com
[9] jelastic.com

Examples of SaaS are Oracle Taleo Cloud Service[10], Netsuite[11], Constant Contact[12], etc.

In this work, we are interested on service-based applications provisioning in Cloud environments mainly in Cloud infrastructure (IaaS) and Cloud platforms (PaaS). Such applications are described according to SOA. In the rest of the section, we define the SOA architecture and two specifications of service-based applications i.e. Business Process Execution Language (BPEL) and Service Component Architecture (SCA).

2.1.2 Service Oriented Architecture (SOA)

Service Oriented Architecture (SOA) is a software architecture and applications design principle that starts with an interface definition and builds the entire application topology as a topology of interfaces, interface implementations and interface calls [12].

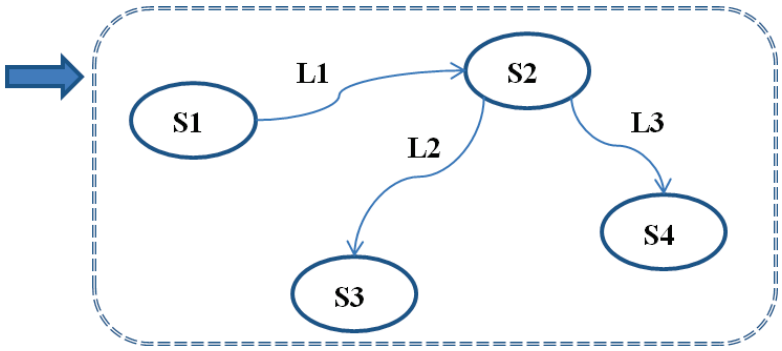


Figure 2.1: Service-based application overview.

SOA covers service-based applications design, description, implementation and invocation. Services are software modules that are accessed by name via an interface, typically in a request-reply mode. Service consumers are software that embeds a service interface proxy (the client representation of the interface) [20]. Service-based application are made of a set of heterogeneous components and appropriate links to ensure interaction between them. The components of such applications can realize service or business process activities and the links to service bindings or call transfers (See Figure 2.1). The linking and orchestration between the different application components are often described in an XML-based descriptor. Service composition specifications can be expressed using Business Process Execution Language (BPEL) or Service Component Architecture (SCA).

[10] http://www.oracle.com/fr/products/applications/taleo/overview/index.html
[11] http://www.netsuite.com/portal/home.shtml
[12] http://www.constantcontact.com/index.jsp

2.1.3 Business Process Execution Language (BPEL)

BPEL is a language and specification to orchestrate Web services into a single business process [15]. BPEL describes the business information exchanges using only the interfaces available through homogeneous services. It includes also such information as when to send messages, when to wait for messages and when to compensate for unsuccessful transactions [21] [22]. Generally, a BPEL process follows the scheme schematized in Figure 2.2.

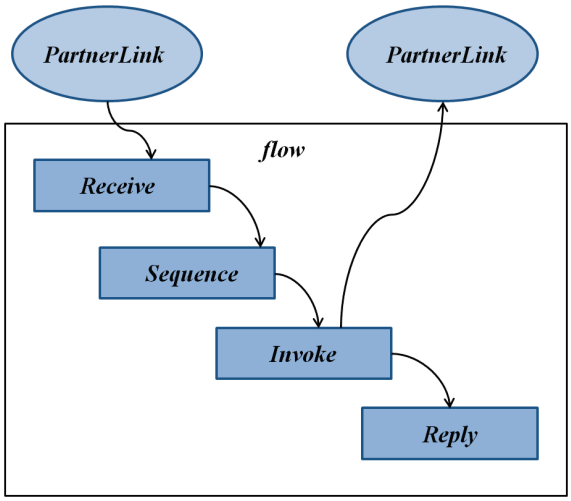


Figure 2.2: Example of BPEL process structure.

A BPEL process can interact and/or invoke external services called *PartnerLinks* through appropriate contracts (e.g. WSDL). The sequencing between the process activities and the logic of the actions is expressed in the XML-based *.bpel* descriptor. Among the main process activities we can cite: *Receive* which senses the request to execute the process, *Assign* which allows to update process variable values, *Invoke* to interact with process' *PartnerLinks*, *Reply* to return the execution result of the process and so on.

As example of BPEL process, we consider the online shop process schematized in Figure 2.3. The correspondent BPEL description of this process is presented in Listing 2.1. When the online shop receives information from a customer, business strategy of the process distinguishes between already known customers and new customers. In case of a known customer, an appropriate branch of the process is executed (Listing 2.1, lines 11-16): first, the shop receives an order (line 13), and then it sends the invoice to the customer (line 14). In case of a new customer (lines 17-28) the shop initiates two tasks concurrently: in the first task (Sequence 1, lines 19-22) the shop first receives the order (line 20) and then confirms it (line 21). In the second task (Sequence 2, lines 23-26) the shop receives the terms of payment (line 24) before it sends an invoice to the customer (line 25). In either case the shop finally sends the

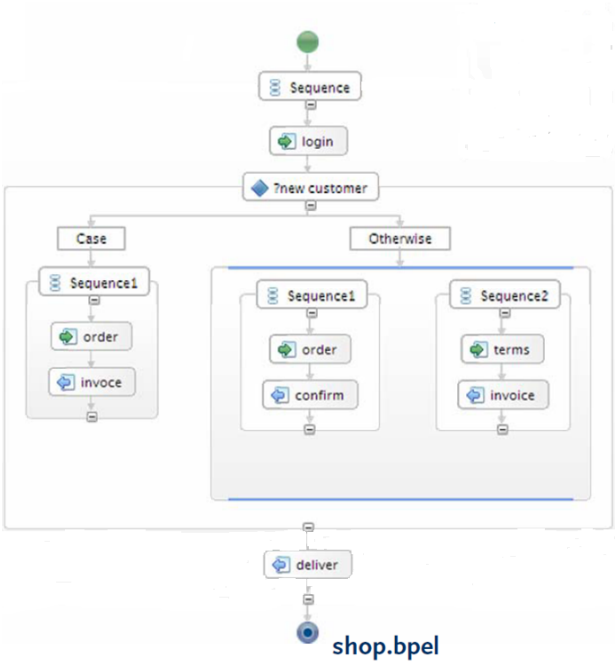


Figure 2.3: The online shop process.

delivery information to the customer (line 30).

Listing 2.1: The online shop process descriptor.

```
1 <process name="Online Shop">
2     <partnerLinks>
3     <partnerLink name="client" partnerLinkType="clientPL"/>
4     </partnerLinks>
5     <variables>
6     <variable name="var" />
7     </variables>
8     <sequence>
9     <receive partnerLink="client" portType="onlineshop" operation="login"
          createInstance="yes" variable="var" />
10    <switch>
11        <case condition="known customer">
12        <sequence>
13            <receive partnerLink="client" portType="onlineshop" operation="
                  order" variable="var" />
14            <invoke partnerLink="client" portType="onlineshop" operation="
                  invoice" inputVariable="var" />
15        </sequence>
16        </case>
17        <otherwise>
18        <flow>
19            <sequence>
20            <receive partnerLink="client" portType="onlineshop" operation="
```

```
                order" variable="var" />
21              <invoke partnerLink="client" portType="onlineshop" operation="
                    confirm" inputVariable="var" />
22              </sequence>
23              <sequence>
24              <receive partnerLink="client" portType="onlineshop" operation="
                    terms" variable="var" />
25              <invoke partnerLink="client" portType="onlineshop" operation="
                    invoice" inputVariable="var"/>
26              </sequence>
27          </flow>
28          </otherwise>
29      </switch>
30      <invoke partnerLink="client" portType="onlineshop" operation="deliver"
            inputVariable="var" />
31       </sequence>
32  </process>
```

2.1.4 Service Component Architecture (SCA)

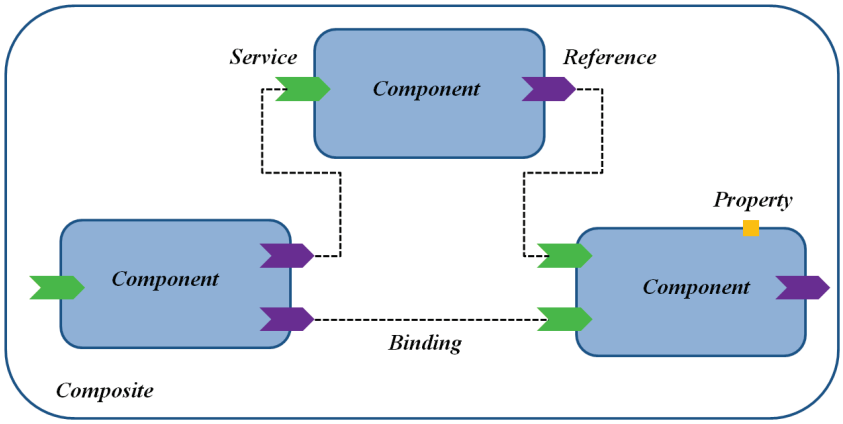


Figure 2.4: SCA-based application architecture.

SCA is a technology and specification for creating services and assembling them into composite applications [23] [13]. In fact, SCA supports a view of services as software components. These components may be implemented using different programing languages. Furthermore, they may require different execution framework and use different communication protocols to interact between them [24] [25]. The architecture of a typical composite is described in Figure 2.4

The *composite* is the main element of an SCA-based application. It consists of a logic box which brings together all of the application *components*. The *components* model the services constituting the application (at least one service per *component*). When they are combined, they interact between them to ensure the business solution of the application. Each *component* offers one or more *services* to the preceding *com-*

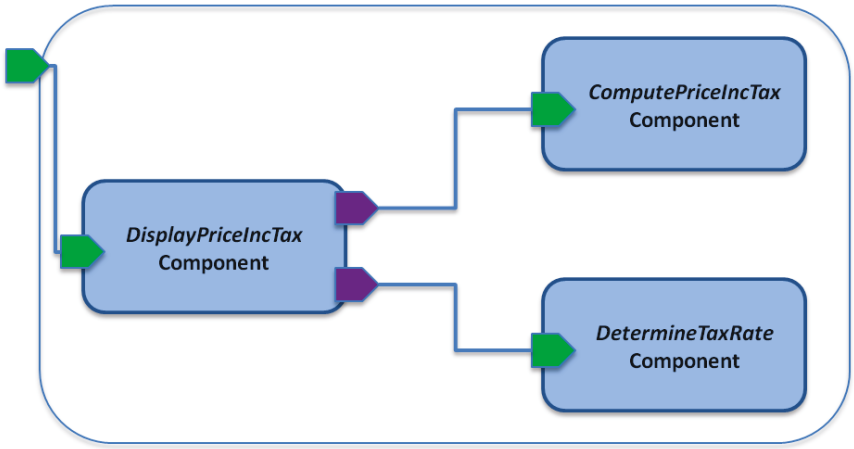


Figure 2.5: ComputePrice SCA-based application.

ponents in the execution chain, provides one or more *references* to the following components and can has one or more *properties*. The link between a *reference* and a *service* is called *binding*. All these elements are described in the XML-based *.composite* descriptor according to the Service Component Description Language (SCDL).

As example of an SCA-based application, we consider an application called computePrice. This application allow to determine a total price (including tax) of a given product based on its provided ID and the tax rate to be applied on it. The application architecture is presented in Figure 2.5 and its correspondent *.composite* descriptor is detailed in Listing 2.2. This application is made up of three components:

- *DisplayPriceIncTax* component,
- *DetermineTaxRate* component,
- *ComputePriceIncTax* component.

The details of the application components are as follows: The *DisplayPriceIncTax* component (Listing. 2.2, lines 3-13) is the front end component of the application, since it is the component ensuring communications with all the other application components through its bindings and ensuring also application interactions with external clients (e.g. receiving queries, responses construction, etc.). This component exposes two references (lines 8-12) to the other application components through two respective bindings: a basic SCA binding with *ComputePriceIncTax* component and a Web Service binding with *DetermineTaxRate* component (line 11). The second component of the application is the *ComputePriceIncTax* component (lines 14-18). This component provides a Java implementation and provides a service that computes the total price of an article based on its basic price and the tax rate communicated by *DisplayPriceIncTax* component. The third component of the application is the *DetermineTaxRate*

component (lines 19-25) which is a remote Web service deployed on a remote Tomcat container instance. This component determines the tax rate from a Database instance based on the product ID provided by *DisplayPriceIncTax* component. It is accessible through an URI (line 23).

Listing 2.2: ComputePrice composite descriptor.

```
1 <?xml version="1.0" encoding="UTF-8"?>
2 <composite  name="ComputePriceApplication">
3   <component name="DisplayPriceIncTaxComponent">
4     <implementation.java class="CalculPrixTtcSCAApplicationWSImplem"/>
5     <service name="CalculPrixTtcSCAService" promote="
          DisplayPriceIncTaxComponent/CalculPrixTtcSCAService">
6       <interface.java interface="CalculPrixTtcSCAServiceInterface"/>
7     </service>
8     <reference name="totalPriceService" target="TtcComponent" promote="
          DisplayPriceIncTaxComponent/totalPriceService">
9     </reference>
10    <reference name="taxService" target="DetermineTaxRateComponent" promote=
          "DisplayPriceIncTaxComponent/taxService">
11      <binding.ws/>
12    </reference>
13  </component>
14  <component name="TtcComponent">
15    <implementation.java class="totalPriceServiceImplem" promote="
          DisplayPriceIncTaxComponent/totalPriceService"/>
16    <reference name="taxService" promote="TtcComponent/taxService">
17    </reference>
18  </component>
19  <component name="DetermineTaxRateComponent">
20    <implementation.java class="taxServiceImplem"/>
21    <service name="taxServiceInterface" promote="
          DisplayPriceIncTaxComponent/taxService">
22      <interface.java interface="taxServiceInterface"/>
23      <binding.ws uri="http://localhost:8080/axis2/taxServiceInterface"/>
24    </service>
25  </component>
26 </composite>
```

2.2 Related work evaluation criteria

To address the SOA support issues mentioned in Section 1.2, Cloud platforms should provide **appropriate mechanisms to support SOA application components heterogeneity**. In addition to that, to enable automatic provisioning of these applications and enhance their portability in such context, we need to use a **generic and common application and Cloud resources description model**. Such generic model unifies the application provisioning and management steps and enhance applications portability since it will be described and provisioned, as well as its resources, in the same way whatever is the target Cloud platform.

In the following, we enumerate and discuss results of related works which are interested in addressing these issues. To evaluate these works, we rely on the following criteria:

- SOA support: Appropriate provisioned Cloud resources to support applications described according to SOA hosting and execution,
- Applications portability: Provided solutions to minimize modifications and adaptations of applications from a Cloud platform to another,
- Standardized resources description model: The proposal or not of common and unified resources description model,
- Standardized user API: The proposal or not of generic user API operations implementing the resources description model.

2.3 Approaches for applications provisioning in the Cloud

In the following, we present and discuss results of related research projects. For each one of these projects, we describe its contributions and performed solutions.

2.3.1 PaaSage

PaaSage[13], a model-based Cloud platform upperware, is an European project that aims designing and implementing a platform for applications development and deployment in the Cloud using an appropriate predefined methodology [2]. PaaSage's model-based methodology supports the Cloud lifecycle phases of configuration, deployment and execution. These phases are based on the Waterfall Model of Software Development [1] illustrated in Figure 2.6.

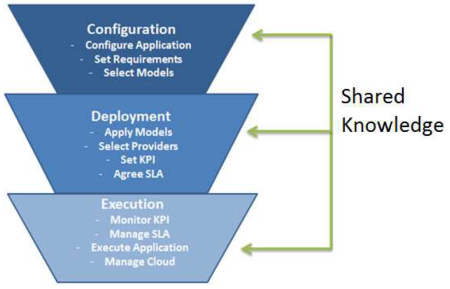


Figure 2.6: Application lifecycle overview [1].

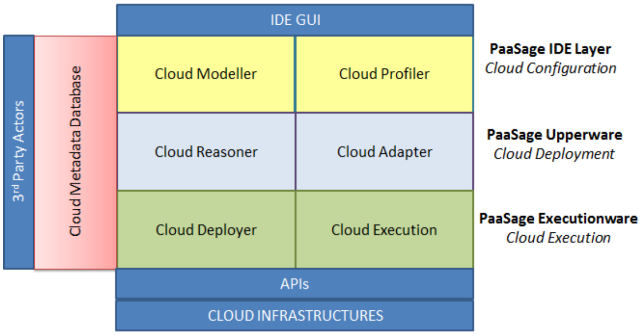


Figure 2.7: Main PaaSage architectural stack [2].

[13] http ://www.paasage.eu/

PaaSage architecture is composed of 3 main layers (See Figure 2.7):

1. *Integrated Development Environment (IDE) layer*: implements the portal of PaaSage platform. It consists of an extended version of the open source development platform Eclipse[14] supporting the chosen Cloud Application Modelling Execution Language (CAMEL) including CloudML [26] [27]. The IDE ensures the model-based integration of the various functional components in the application project.

2. *Upperware layer*: provides a set of tools for application requirements express at design time. The *Cloud Reasoner* allows the use of PaaSage model-based knowledge integrated to the IDE and ensures mediation between the IDE layer and the *Executionware* layer.

3. *Executionware layer*: provides platform-specific mapping and technical integration of PaaSage to the provider APIs. The *Deployer* ensures the application components deployment while the *Adapter* interacts with the target provider API.

PaaSage IDE allow development of applications described according to SOA through appropriate plugins. The deployment and the execution of these application is performed in its own platform (*Executionware layer*). Archiving application is handled by the *Cloud Adapter* in a specific way on the *upperware layer* before the deployment which hampers application portability. PaaSage provides no standardization effort for resources description and user API. It uses proprietary Cloud infrastructure APIs when allocating resources.

Table 2.1: PaaSage project synthesis.

SOA support	Portability	Standardized model	Standardized API
YES	NO	NO	NO

2.3.2 mOSAIC project

mOSAIC[15], result of an European collaborative project, is an open-source API and platform for multiple Cloud systems. mOSAIC offer tools for developing portable Cloud-applications which can consume hardware and software resources offered by multiple Cloud providers [3].

Figure 2.8 details the mOSAIC platform architecture. The *Application Support* includes the *API implementations* and *Application Tools*, as well as the *Semantic Engine* and *Service Discoverer*. The API exposes operations to build and manage

[14] http://www.eclipse.org/
[15] www.mosaic-cloud.eu

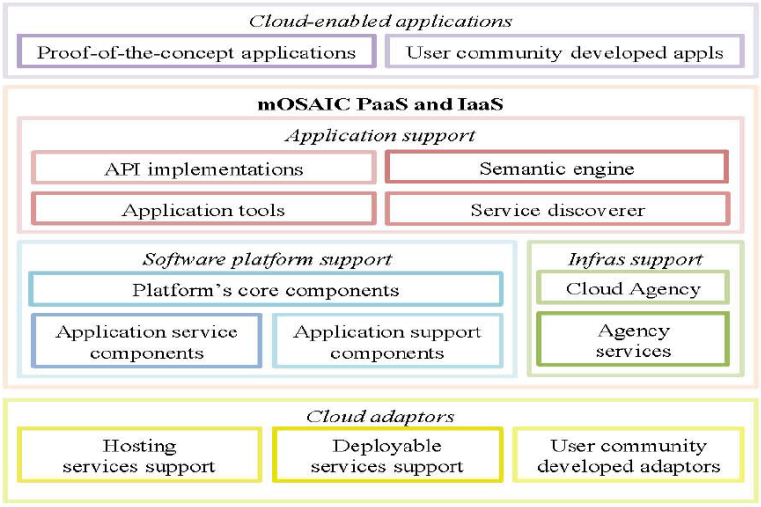


Figure 2.8: mOSAIC platform overview [3].

Cloudlets. A Cloudlet represents an abstraction of application functionnality and is subject to be elastic and monitored by the *Sotware Platform*. The *Semantic Engine* is the sub-system supporting the user in selecting APIs components and functionalities needed to build the Cloudlet, and resources to be allocated from the Cloud providers. The API operations are generic. This ensures a degree of abstraction of *Software Platform Support* when describing the application and its resources.

The *Software Platform Support* manages the hosting environment of the application to deploy based on deployment and application descriptors. The *Software Platform Support* identifies the application components and the needed Cloud resources when parsing the application descriptor. Provided resources do not support appropriate frameworks and engines for applications described according to SOA execution. However, the *Software Platform Support* is open source and could be extended by the community to support SOA in the near future.

The *Infrastructure Support* provides concrete Cloud resources and services to be provisioned. The *Cloud Agency* performs required service providers provisioning through *Vendor agents* (e.g. storage service). The selection of the target provider is based on the brokerage contracts. The *Cloud Adaptors* implement the specific provider operations to perform the provisioning. Currently, mOSAIC provide *Adaptors* for Amazon EC2, Flexiscale, Eucalyptus and OpenNebula.

Development and deployment stages in mOSAIC are treated separately to avoid portability issues. Indeed, development is done independently of the Cloud platform while deployment consists in selecting required Cloud resources and starting the application components. Before deployment time, the developer describes the application components' requirements and dependencies in terms of communication and data in a unified way through the API implementations.

Table 2.2: mOSAIC project synthesis.

SOA support	Portability	Standardized model	Standardized API
NO	YES	YES	YES

2.3.3 Cloud4SOA

Cloud4SOA[16] is an European research project that provides an interoperable multi-PaaS Cloud solution for SOA applications. Cloud4SOA offers to developers the ability to select, deploy and manage applications in several PaaS [4]. The main contributions of this project consists of providing facilities to developers to switch their applications from one PaaS to another with minimum change and adpatation efforts, so Cloud4SOA aims at enabling applications portability. Cloud4SOA establish an abstration layer, based on an appropriate defined ontology, upstream different PaaS and exposes a common generic management interface to ensure the interoperability of the solution.

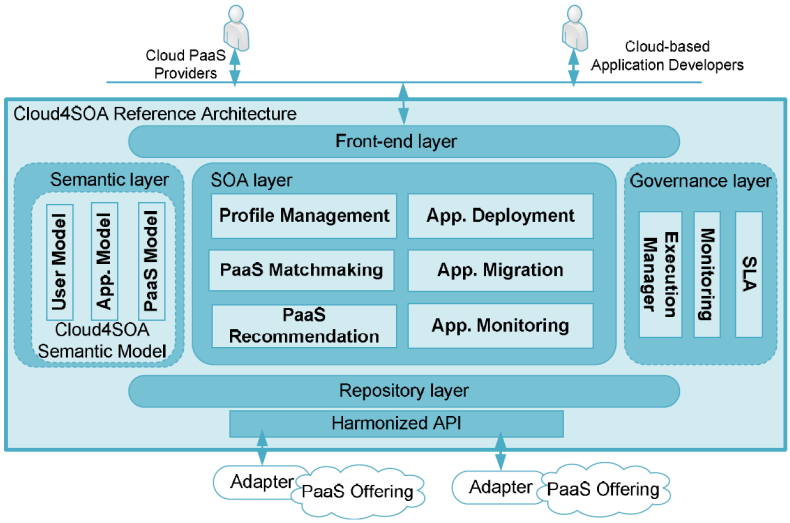


Figure 2.9: Cloud4SOA reference architecture [4].

Figure 2.9 details the reference architecure introduced by Cloud4SOA. This architecture consists of 5 layers:

- *Service Front-end layer*: provides a dashboard and a set of GUIs to access to the Cloud4SOA features
- *Semantic layer*: provides the formal representation of information (e.g. PaaS resources, application dependencies, etc.). The 3 parts of the ecosystem are supported by this fomalization (i.e. the Cloud end-user, the Cloud-based application and the Cloud PaaS provider).

[16] www.cloud4soa.eu/

- *SOA layer*: provides the Cloud4SOA's core functionalities. This layer includes the *Profile Management* module responsible of managing the PaaS offerings, applications and user profiles. The *PaaS Matchmaking* module selects the best PaaS offerings based on information of the *Repository layer* and user requirements. The *PaaS Recommendation* module calculates similarities between an application and a PaaS offering. These similarities and SLA violations events allow to rate available PaaS offerings and intervenes in the matchmaking procedure. The *Application Deployment* module processes the applications deployment and governance (start, stop and undeploy) in a PaaS. The *Application Deployment* module process deployment on appropriate allocated resources from connected PaaS offerings through a harmonized API. The *Application Migration* module processes migration of applications from one PaaS to another. The *Application Monitoring* module provides an interface to access to the monitoring functionality and to retrieve the collected data according to different parameters.

- *Governance layer*: provides implementation of the business-centric focus of Cloud4SOA. It allows management of applications lifecycle. The *Execution Manager* performs applications (un)deployment, migration and maintenance. The *Monitoring* module performs monitoring of deployed applications. The SLA-based services are responsible of specific treatements related to user requirements (e.g. supervising SLA guarantees, SLA violations dealing, etc.).

- *Repository layer*: provides records as RDF triples related to developer's profiles and supported PaaS providers' capabilities.

The provisioning process is performed through a harmonized API that exposes generic application management operations. A specific adapters ensure the mapping between the harmonized API and the proprietary PaaS offering APIs.

Table 2.3: Cloud4SOA project synthesis.

SOA support	Portability	Standardized model	Standardized API
YES/NO*	YES	YES	YES

(*) Cloud4SOA provides mechanisms to provision applications described according to SOA in connected PaaS offerings as long as they support SOA applications hosting and execution.

2.3.4 Contrail Project

The Contrail[17] project aims to design, implement, evaluate and promote an open source computational Cloud wherein users can limitlessly share resources [28]. The Contrail vision is a federation of resources provided by public and private Clouds.

[17] contrail-project.eu

Offered Cloud resources should be integrated into a single homogeneous federated Cloud that users can access seamlessly. Any organization should be able to be both a Cloud provider, when its IT infrastructure is not used at its maximal capacity, and a Cloud customer in periods of peak activity. Infrastructure and platform introduced by the Contrail project are introduced in Figure 2.10.

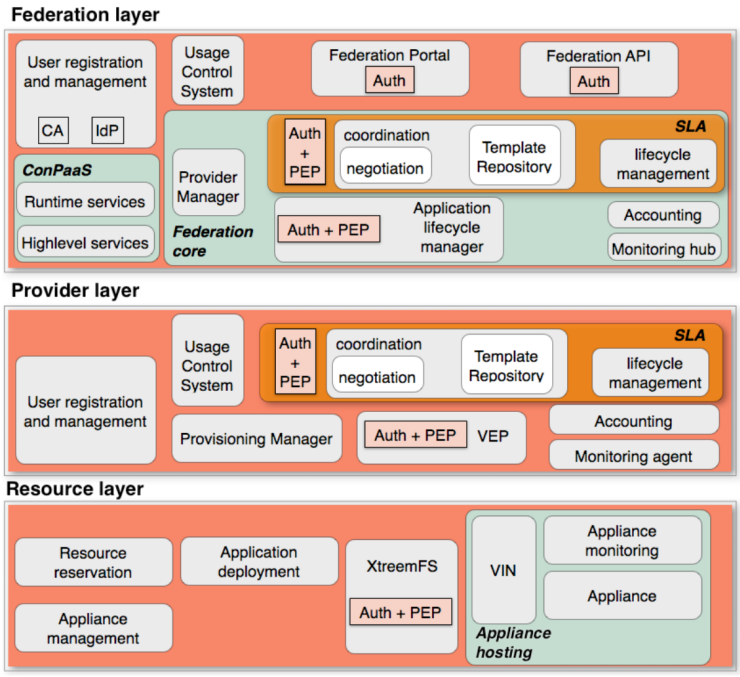


Figure 2.10: Contrail architecure [5].

This architecture consists of 3 layers:

- *Federation layer* which implements a broker-based solution to manage and negotiate contracts with the providers based on Service Level Agreements (SLA) documents. This layer exposes a REST API to manage applications, data, users and providers,
- *Provider layer* which manages applications and services provisioning and execution according to the provided SLA (e.g. ConPaaS [29]).
- *Resource layer* which aggregates resource characteristics and capabilities.

The Contrail project does not support deployment of applications described according to SOA. The applications deployment mechanism is based on the Open Virtualization Format (OVF). OVF is an open standard for packaging and distributing software to be deployed and run in virtual machines. This facilitates the import/-export applications from one provider to another. However, this requires that the

target provider supports OVF processing. The federation API exposes aggregated management operations common to providers. They enable managing applications and resources in the same way when even if there are delivered by different providers.

Table 2.4: Contrail project synthesis.

SOA support	Portability	Standardized model	Standardized API
NO	NO	YES	YES

2.3.5 Other research projects

The Cloud-TM project[18] proposes a middleware platform which exposes a set of APIs and abstractions for the development, provisioning and administration of large scale applications across a dynamic set of distributed nodes allocated from IaaS Cloud providers [30].

4CaaSt[19] proposes a solution ensuring the development, description and deployment of applications [31]. The three processes are seperated. The user can develop and describe the application through 4CaaSt before delegating the hosting environment selection and the deployment to the platform. Application archives need to be adapted after the selection of the hosting platform.

The RESERVOIR[20] project provides an infrastructure that allows reliable services provisioning. Connected Cloud providers to RESERVOIR address end-users' requirements leasing computational resources from IaaS providers which interoperate with each other creating a seamlessly pool of resources and address interoperability issues based on an OCCI infrastructure implementation [32].

Table 2.5: Other project syntheses.

Project	SOA support	Portability	Standardized model	Standardized API
Cloud-TM	NO	NO	YES	YES
4CaaSt	NO	NO	YES	YES
RESERVOIR	NO	NO	YES	YES

2.3.6 Related academic work

When considering generalist application architectures, provisioning procedures are basically based on resource allocation algorithms that match between applications requirements, provided by the end user, and available node resources at the provider side in order to maximize the performances and minimize the operating cost [33] [34] [35].

[18] www.cloudtm.eu/
[19] www.4caast.eu/
[20] www.reservoir-fp7.eu/

When considering service-based applications, there are a lot of work which focused on provisioning appropriate platform resources to support their hosting and execution.

For example, for business processes, the authors in [36] draw up an inventory of the different delivery models available to execute a BPEL process at IaaS, PaaS and SaaS layers. The conclusion made by the authors stipulates that there is still several lacunas to support BPEL processes provisioning in the Cloud especially in terms of communication processing between the different activities of the process.

In [37], the authors propose a context-oriented methodology supporting Cloud workflows design. This approach interacts strongly with the user to allow him to make contexts explicit in designing Cloud workflow models. The users could then customize information, formalize their design strategies, and possibly interact with the system in a collaborative pattern in order to perform Cloud workflows operating.

Moreover, some works have proposed to transform and slice business processes to equivalent sub-processes in order to be able to deploy them (or a part of them) in the Cloud [36] [38] [39].

Other approaches consists in defining algorithms to place and execute workflows over allocated Cloud resources optimally [40] [41]. These approaches consider the data dependencies between workflow steps and the utilization of resources at runtime to place the process components over these resources. Placement and resources allocation decisions are based on predefined heuristics. Provided implementations are based on the ActiveBPEL engine for processes executing and Amazon's Elastic Compute Cloud for required resources allocation.

In addition to that, it should be noted that several PaaS providers begin to integrate features to design, deploy and execute business processes in the Cloud. For example, Amazon Web Service proposes the Amazon Simple Workflow Service (Amazon SWF) which assists developers to coordinate the various processing steps in the process to deploy and allow them to manage distributed execution state [42]. Salesforce introduced a tool called "Visual Process Manager" to support business process management on Salesforce platform [43]. WSO2, an Apache ODE-based process engine, provides a variant of its business process server as-a-Service [44].

Regarding applications based on services compositions, some works have focused on defining new PaaS prototypes dedicated to support such applications and provision their needed hosting frameworks. For example, in [35], a PaaS prototype relies on a configurable kernel which is inspired from FRASCATI[21], an open source implementation of SCA specifications, to support SCA runtime. There are also additional works based on FRASCATI and supports reconfiguration of SCA applications from the domain of Software Product Line (SPL) design provided by the developer [45] [46]. Furthermore, FRASCATI was integrated as standalone framework over several existing PaaS providers to support execution over these PaaS (e.g. FRASCATI in

[21] http://wiki.ow2.org/frascati/Wiki.jsp?page=FraSCAti

Google App Engine[22], FRASCATI in Heroku[23], etc.)

In [47], the authors introduced an approach to deploy SCA applications in Cloud environments. In this approach, the initial application provided by the developer is an SOA application described through a BPMN diagram. This latter is transformed to a basic equivalent SCA view using Mangrove core tool. After that, the developer has the ability to refine the application architecture (e.g. introducing additional components, defining external dependencies, etc.) and augment then the application components with virtual node (VN) names. VN concept is an abstraction for the deployment of distributed applications used by GCM/ProActive wich represents an implementation of the Grid Component Model (GCM) [48]. The VN abstraction is used to refer to the location where the GCM components will be deployed without actually specifying physical nodes, and delaying this association to the moment when the actual resources are available. From this augmented description, an equivalent GCM architecture description language descriptor is generated. This descriptor contains the GCM components, their bindings and the VNs where they will be deployed, thus offering a vision obtained directly from the architectural design, and at the same time closer to infrastructure concerns [47].

In addition to that, the Apache Foundation tried to develop a Cloud-aware version of Apache Tuscany to be able to be hosted and provisioned efficiently by a PaaS provider [49]. They also start a new project called Apache Nuvem which aims to define a novel API for Cloud application services, to support SOA applications (SCA-based applications included) deployment across the most popular Cloud providers [50].

2.4 Approaches for Cloud resources description

In the following, we discuss existing related work regarding platform and application resources description models.

2.4.1 Topology and Orchestration Specification for Cloud Applications (TOSCA)

TOSCA is a specification, supported by OASIS, that provides an XML-based language to describe PaaS applications as a set of *Nodes* with well defined *Relationships* [6]. The *Nodes* and *Relationships* are described in *Topology Templates* which are schematized as a typed graph (See Figure 2.11). Such templates provide the structure of the application to deploy and the needed details to set its hosting environment and its artifacts. A *Node* is instantiated from a *Node Template* specifying the occurrence of a *Node Type* as a component of the application. A *Node Type* defines the properties of such a component (*properties*) and the operations (*interfaces*) available to manipulate the component.

[22] http://ow2-frascati.appspot.com/
[23] http://frascati.herokuapp.com/

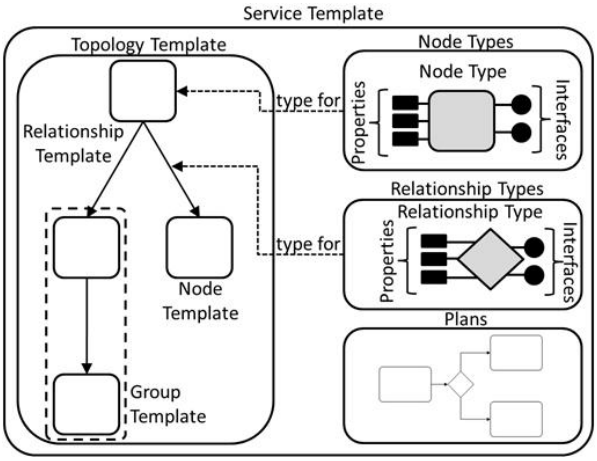


Figure 2.11: TOSCA Service Template overview [6].

A *Relationship* is instantiated from *Relationship Template* specifying the occurrence of a relationship between *Nodes*. Each *Relationship Template* refers to a *Relationship Type* that defines the semantics and properties of the *Relationship*. The *Relationship Type* indicates the elements it connects and the direction of the relationship (source and target elements) and optionally constraints (if needed) to satisfy in order to establish the relationship. Note that, *Relationship Types* (respectively *Node Types*) are defined separately, so they enable reuse perspectives.

The *Service Templates* contains also a *Plan* element that describes the process models used to create the application and its needed operational management behavior once deployed (e.g. scaling, backuping, etc.). The *Plans* are instantiated from *Plan Models* and are implemented as workflows (e.g. BPMN, BPEL, etc.) to benefit from compensation, recovery, and transaction concepts [51]. *Plans* provide values for the various *Node Templates* and *Relationship Templates* properties.

Concretely, according to TOSCA specifications, a deployed application is an instance of a *Service Template*. This instance is created by instantiating the *Topology Template* of its *Service Template*. This instantiation is performed by running a dedicated *Plan* defined for the *Service Template*. An appropriate *CSAR* might be used to package the *Node* and the *Relationship* reperesenting this application.

For deployment, TOSCA application elements are encapsulated in a predefined archive format called *Cloud Service ARchive (CSAR)*. *CSAR* is composed at least of two directories. The first directory is the TOSCA meta file (*.meta)* which describes metadata of all other files in the *CSAR*. The second one is the definition directory which contains typically sources and definitions related to the application (*.tosca)*. The use of *CSAR* meet with applications described according to SOA specifications but the *Service Template* notion hampers the application portability. Indeed, this requires a set of modification and adaptation in the PaaS side. Each Cloud platform would

map the specified *Service Topology* to its available concrete infrastructure in order to support concrete instances of the application and adapt the management plans accordingly. In addition to that, most of existing PaaS do not support currently *CSAR* deployment even if there is some platform prototypes supporting TOSCA specifications (e.g. ServIce Offering and Provisioning Platform - SIOPP[24][52], OpenTOSCA [53]).

Table 2.6: TOSCA synthesis.

SOA support	Portability	Standardized model	Standardized API
YES	NO	YES	YES

2.4.2 Cloud Application Management for Platforms (CAMP)

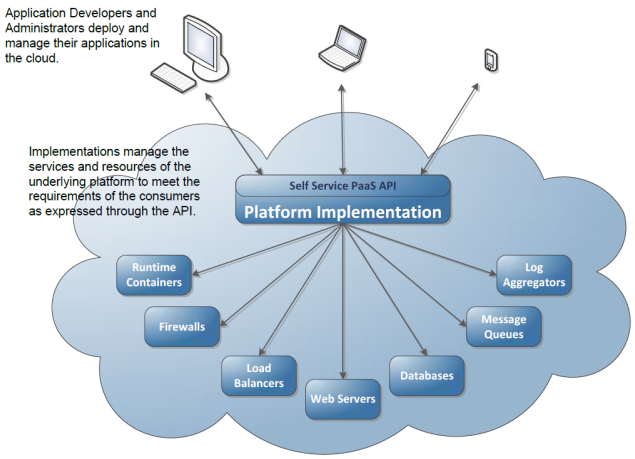


Figure 2.12: Typical PaaS architecture [7].

Cloud Application Management for Platforms (CAMP) is a specification and REST API standardized by OASIS [7]. CAMP provides basis for developing multi-cloud management tools as well as a REST-based approach to application management across public and private Cloud computing platforms. Provided management operations involve application description, packaging and deployment.

Before defining application operations management and PaaS API specifications, CAMP performed a census for existing PaaS resources. This is useful besause these are the resources that will ensure and execute the operations exposed by CAMP API on the provider side. Figure 2.12 schematizes a typical PaaS architecture as it is defined by CAMP. The PaaS provider implements a set of technical components in order to meet the deployed application requirements. These requirements are exposed

[24]SIOPP is pronounced 'shop'

through the self service API operations by the application developers. The details of identified platform components are as follows:

- *Runtime containers* to provide runtime environment and execution frameworks to applications,
- *Firewalls* to secure applications invoking,
- *Load Balancers* and *Message Queues* to manage applications instances and received requests,
- *Web Servers* to host and execute Web applications,
- *Databases* to manage data persistence,
- *Log aggregators* for applications logging.

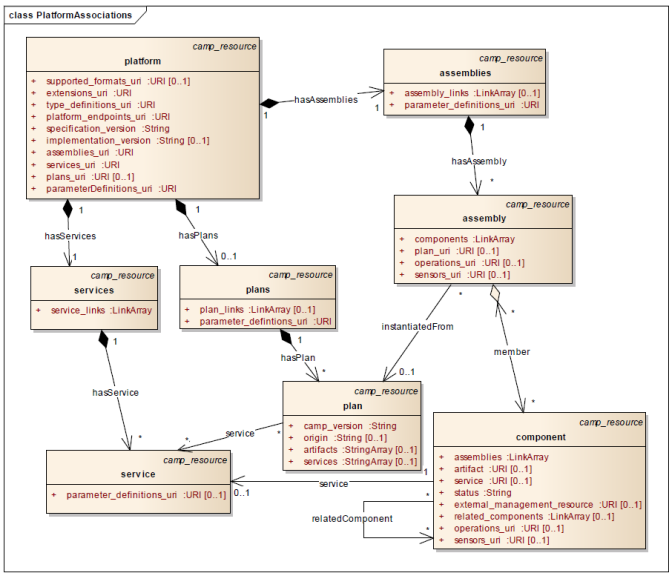


Figure 2.13: CAMP platform resources and relationships [7].

Based on this census, CAMP defines specifications to formalize applications, PaaS resources and relations between them (See Figure 2.13). CAMP represents the PaaS as a set of related application components and using *Platform* components via an *Assembly* resource. *Assembly* resources represent a running application. The *Platform* resource is described as a set of platform *Components* offering a list of capabilities to be used by applications. Similarly, an application is a set of application *Components* having capabilities and requirements. An application *Component* can be related to a platform *Component* if this latter has the needed capabilities that could be associated to the requirements of the application *Component*.

CAMP description model standardizes application provisioning and management processes. In fact, the technology vendors behind the CAMP specifications, which include CloudBees, Cloudsoft Corporation, Huawei, Oracle, Rackspace, Red Hat and Software AG, is based on the same common management operations for applications. The applications are encapsulated is specific packages called Platform Deployment Package (PDP). The PDP archive contains a *Plan* file and the application content files (e.g. sources code, scripts, etc.). The *Plan* file describes the application *Assembly(ies)* and optionally needed *Service(s)*. The *Services* are provisioned by the PaaS to satisfy the application requirements and dependencies (e.g. load balancing, database linking, etc.). The PDP format is specific and requires adaptation on the PaaS side to support PDP archives and plan documents (YAML-based documents). Moreover, the PDP allow describing applications described according to SOA. However, the application lifecycle management provided by CAMP is still Web-oriented and is not really appropriate for service-based applications.

Table 2.7: CAMP synthesis.

SOA support	Portability	Standardized model	Standardized API
YES	NO	YES	YES

2.4.3 Open Cloud Computing Interface (OCCI)

OCCI is a set of specifications that defines a meta-model for abstract Cloud resources and a RESTful protocol for their management. It offers a flexible API with a strong focus on interoperability while still offering a high degree of extensibility.

To enhance modularity and extensibility, OCCI is released as a suite of complimentary documents such as:

1. OCCI core that defines a meta-model for Cloud resources description and management [8],
2. OCCI rendering specifications that contains multiple documents describing rendering of the OCCI core model [54],
3. OCCI extensions which are instantiations of the OCCI core meta-model to model particular Cloud resources (e.g. infrastructure resources [9]).

Figure 2.14 schematizes defined types of the core model, as it was introduced in [8], and relations between them. The *Kind* type is the core of the types classification system built into the OCCI core model. *Kind* is a sort of specialization of *Category*. It allows definition of all resource capabilities in terms of management actions. An *Action* represents an invocable operation applicable to a *Resource* instance. Any resource exposed through OCCI is a *Resource* or a sub-type of *Resource* instance. The *Resource* type is complemented by the *Link* type which associates one *Resource* instance to an

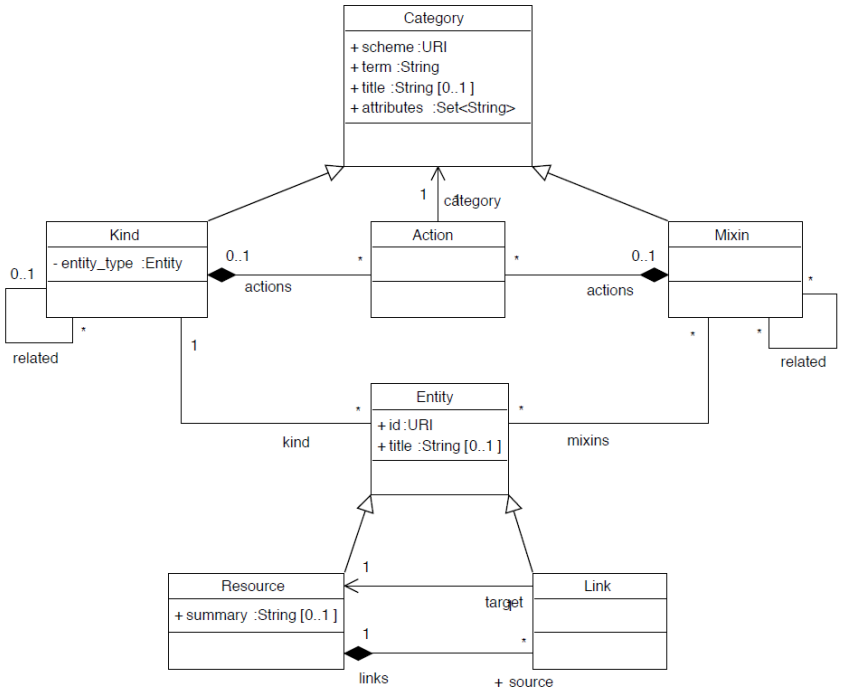


Figure 2.14: UML class diagram of the OCCI core model [8].

another. *Entity* is an abstract type, which both *Resource* and *Link* inherit. Each sub-type of *Entity* is identified by a unique *Kind* instance. Finally, an instance of *Mixin* type can be associated with a resource instance, i.e. a sub-type of *Entity*, to mix-in additional resource capabilities at run-time.

The OCCI core model is suitable to be extended and serve many other resource description models such as infrastructure resources. Figure 2.15 illustrates defined types for the OCCI infrastructure model as it was introduced in [9]. These infrastructure types inherit the OCCI core model *Resource* base type and all their attributes. The HTTP Rendering document defines how to serialize and interact with these types using RESTful communication. Table 2.8 lists the OCCI infrastructure types and their related links. *Compute*, *Storage* and *Network* types inherit the *Resource* base type defined in OCCI core model. They represent respectively a generic information processing resource (e.g. virtual machine, CPU), networking devices (e.g. switch) and data storage devices (e.g. disk). The *StorageLink* type represents a link from a *Resource* to a target *Storage* instance (e.g. Linking a VM to a disk) while the *NetworkInterface* allow interacting with a *Network* instance (e.g. network adapter). It can extended using the mix-in mechanism to support specific capabilities (e.g. *Ipnetworking* mixin for TCP/IP capabilities). In addition to that, the extension provides a set of attributes to well describe the defined types and a set of operations to manage

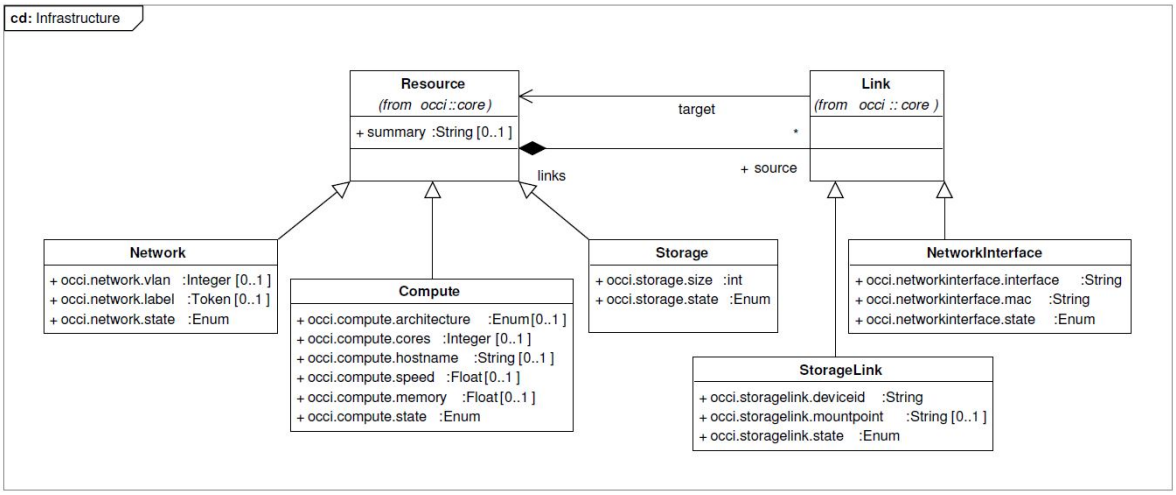


Figure 2.15: Overview diagram of OCCI infrastructure types [9].

Table 2.8: The kind instances defined for the infrastructure subtypes of Resources, Links and related Mixins.

Term	Scheme	Related Kind
Compute	$< schema > /infrastructure\#$	$< schema > /core\#resource$
Storage	$< schema > /infrastructure\#$	$< schema > /core\#resource$
Network	$< schema > /infrastructure\#$	$< schema > /core\#resource$
StorageLink	$< schema > /infrastructure\#$	$< schema > /core\#link$
NetworkInterface	$< schema > /infrastructure\#$	$< schema > /core\#link$
Ipnetworking	$< schema > /infrastructure/network\#$	-

them (e.g. start, stop, etc.).

To summarize, OCCI entities are abstract and need to be extended to support SOA applications. In addition to that, OCCI specifications and API are generic and are intented to unify Cloud resources provisioning and management. This standard is open source, so they allow interoperability and portability of resources applications included.

Table 2.9: OCCI synthesis.

SOA support	Portability	Standardized model	Standardized API
NO	YES	YES	YES

2.4.4 Related academic work

In the literature, we found several works that tried to define generic operations for Cloud provider resources management. These works attempt to benefit from identified similarities between Cloud platform representations. In [55], the authors discuss the need for a generic API that enables Cloud users to specify their requirements among provider offers. Their investigations shows that most of these APIs use similar concepts with similar properties and actions but with different names and structures. The authors consider that the interoperability and portability problems arise due to different modeling and notation of the same features across different Cloud providers. To handle this issue, the author considers semantic technologies as a solution for interoperability and portability in the Cloud.

In [56], the authors propose to use existing approaches to describe applications and their deployment procedure in the Cloud. They use resource templates (as for TOSCA and CloudFormation) representing reconfigurable entities that can be reused for different applications. Automated deployment of the resources associated to templates description, can be possible using deployment recipes (using DevOps technologies like Chef[25] or Puppet[26])

In [4], the authors suggest that *a common API should involve a set of core functionalities that will meet the basic needs of any Cloud Platform and will unify all different APIs (an API for all APIs)*. Accordingly, a common API is proposed with a common semantics for the needed PaaS resources and actions. This API communicates with PaaS providers via adapters, and any new provider has to adapt his offering following the same semantics (i.e. using the same models and structures or providing an adapter to transform its own representation to the common one). To deploy an application, developers should provide an application profile that describes the requirements of the application. A management module builds an application deployment descriptor according to the selected PaaS, and then, initiates the application deployment via a standard API (i.e. Cloud4SOA [57]) that uses the dedicated adapter for the selected PaaS offer.

2.5 Synthesis

Table 2.10 details a synthesis of presented related work results. The cited research projects propose concrete approaches for services and applications provisioning in Cloud providers. Some of them provide solutions to address portability issues in order to enable multi-cloud deployment and address portability and vendor lock-in issues (e.g. mOSAIC, Contrail). However, the majority of them failed to provide efficient mechanisms to support the high heterogeneity of applications described according to SOA. For example, Cloud4SOA allow the deployment of service-based applications

[25] www.getchef.com/chef/
[26] puppetlabs.com/

in connected PaaS offerings if the target PaaS supports provisioning of its required resources.

Table 2.10: Synthesis of related work results.

Project	SOA support	Portability	Standardized model	Standardized API
mOSAIC	NO	YES	YES	YES
PaaSage	YES	NO	NO	NO
Cloud4SOA	YES/NO	YES	YES	YES
Contrail	NO	NO	YES	YES
Cloud-TM	NO	NO	YES	YES
4CaaSt	NO	NO	YES	YES
RESERVOIR	NO	NO	YES	YES
TOSCA	YES	NO	YES	YES
CAMP	YES	NO	YES	YES
OCCI	NO	YES	YES	YES

For the cited academic and existing related work, we noticed that approaches which are based on the integration of classical service containers in Cloud environment have limitations. These features are extensions of the Cloud system predefined resources rather than the support of service-based applications execution and hosting (e.g. FRASCATI-based PaaS solutions, Apache Tuscany cloudware, etc.). Indeed, based on experimentations results that we have conducted (See Section 6.3.1), we demonstrated that classic service containers design is still neither elastic nor scalable [58] [59].

TOSCA and CAMP are promizing tentatives of standardization, but their introduced specific application archives format (i.e. *PDP* for CAMP, *CSAR* for TOSCA) penalizes the application portability. Indeed, to support these formats, we need to use specific implementations (e.g. OpenTOSCA) or perform changes on the Cloud platform side (e.g. CloudBees to support CAMP specifications).

To allow portability between end users and existing Cloud platforms, we need to design a description model which would help to make abstraction of any provisioning system, of any Cloud service and/or resource. Such a model would enable a detailed description of complex workloads in order to provision them in an automated fashion on heterogeneous providers. To achieve this goal, we believe that we need an object-oriented model specifically designed for Cloud Computing and flexible enough to be enriched with new extensions. For all of these reasons, we choose the OGF OCCI open standard. Indeed, to the best of our knowledge, OCCI is the only standard model compliant with these criteria. Moreover, OCCI is an open standard, that in addition to ensure interoperability and portability, allows not loosing the efficiency and specificities of the Cloud systems. One of the objectives that we want to achieve in this thesis is to extend OCCI in order to define unified platform and application resources description model and to propose a REST API implementing this model and allowing provisioning such resources in a unified way whatever is the

target PaaS. Such unification upstream Cloud platforms facilitates cooperation and federation between them and probably will convince users who are still hesitant to adopt the PaaS economic model. A lot of works have already dealt with this aspect at IaaS layer and has contribute to cooperate and federate data centers and infrastructure resources [60] [32] [61]. Furthermore, the resources introduced by OCCI extension should support applications described according to SOA hosting and execution.

2.6 Conclusion

In this Chapter, we highlighted Cloud platform limitations and drawbacks related to service-based applications provisioning. We presented a set of related collaborative research projects that attempt to address these issues. We also detailed a set of standardization tentatives of application and platform resources description that aims to unify provisioning and management operations. In the last Section of the Chapter, we draw a synthesis and we discuss results of cited works.

Chapter 3

Service-based Application Slicing

Contents

3.1 Introduction

Our defined SPD approach to provision a service-based application in Cloud environments consists of 3 steps: (1) Slicing the service-based application into a set of elementary and autonomous services, (2) Packaging these services in micro-containers and (3) Deploying the micro-containers in the target Cloud environment [62]. In this Chapter, we present and details the first step of this SPD approach.

Since service-based applications have often highly heterogeneous components (See Section 1.1), it is difficult to satisfy its requirements by provisioning one allocated hosting environment based on existing Cloud environment capacities. Because of this, we propose to slice the applications into a set of elementary and autonomous services before allocating a dedicated and appropriate environment for each one of the obtained

services. Moreover, the slicing facilitates the deployment of the applications and allows us to susbstitue the initial application components orchestrations by services choreographies when executing the application.

For service-based applications, it is not possible to determine in advance the execution branch that it will follow at runtime. Indeed, branch execution choices depend on several unpredictable parameters criteria (e.g. variable and parameter values, human interactions in the middle of the process, handlers processing, etc.). All these aspects should then be taken into consideration in the slicing step. Furthermore, since these applications are various in terms of execution specifications (e.g. services choreography for some, services orchestration for others, etc.) and correspondent descriptors does not use the same description languages (e.g. Service Component Description Language (SCDL), etc.), we perform appropriate slicing algorithms. Broadly speaking, this step aims at slicing applications described according to SOA based on formal representation of their services compositions. We handles in this step slicing of:

- Applications modeled as business processes,
- Applications modeled as compositions of service components.

To perform slicing, we chose to formally represent the service-based applications, and then process the slicing. This allows us to keep an eye on sliced services interaction and orchestration with other services and then facilitate the verification procedure of the semantics preservation. Specifically, for applications modeled as business processes, we covers all specifications that can be modelized on Petri nets such as BPEL or BPMN processes. For applications modeled as compositions of services, we cover all specifications whose services compositions can be modelized as graph-based composition (directed graph) such SCA-based applications.

This Chapter is organized as follows: We present our defined algorithms to slice business processes in Section 3.2. We present our defined algorithm to slice applications modeled as compositions of service components in Section 3.3. Proof of preservation of processes semantic and illustrative examples are provided for both cases.

3.2 Slicing of business processes

For business processes slicing, we opted for the use of Petri nets to formally describe the processes and then, proceed to the slicing step directly on the correspondent Petri net graphs. The use of Petri net representation as an intermediate step allows us to preserve semantics of sliced processes.

Preliminaries on Petri nets and WF-nets are firstly introduced in Section 3.2.1. The slicing algorithms are presented in Section 3.2.2. Addressed algorithms enable business processes decomposition into a set of dependent WF-nets through an intermediate Petri net representation. A function to determine the dependencies between the obtained WF-nets is carried. Based on the results of this function and applying

the theorem we have defined, we retrieve the same execution of the initial business process. The proof of the defined theorem is given in Section 3.2.3. The algorithms that we have defined are not costly when operating the business processes as they are executed only once before deploying the processes. To illustrate our findings, we provide an example of a BPEL process slicing in Section 3.2.4.

3.2.1 Preliminaries: Petri nets, WF-nets

In this following, we present some preliminary notions on Petri nets.

3.2.1.1 Petri nets

Definition 1. *A Petri net (Place-Transition net) is a bipartie directed graph* $N = \langle P, T, F, W\rangle$ *where:*

- P is a finite set of places (cercles) and T a finite set of transitions (squares) with $(P \cup T) \neq \emptyset$ and $P \cap T = \emptyset$,
- *A flow relation* $F \subseteq (P \times T) \cup (T \times P)$,
- $W : F \to \mathrm{N}^+$ *is a mapping that assigns a positive weight to any arc.*

Each node $x \in P \cup T$ of the net has a pre-set and a post-set defined respectively as follows: ${}^\bullet x = \{y \in P \cup T \mid (y, x) \in F\}$, and $x^\bullet = \{y \in P \cup T \mid (x, y) \in F\}$. Adjacent nodes are then denoted by ${}^\bullet x^\bullet = {}^\bullet x \cup x^\bullet$. Given a set of nodes S, $\mid S \mid$ denotes the cardinality of S i.e. (the number of elements belonging to S).

The incidence matrix C associated with the net is defined as follows : $\forall (p, t) \in P \times T : C(p, t) = W(t, p) - W(p, t)$. A marking of a Petri net N is a function $m : P \to \mathbb{N}$. The initial marking of N is denoted by m_0. The pair $\langle N, m_0\rangle$ is called a Petri net system. A transition t is said to be enabled by a marking m (denoted by $m \xrightarrow{t}$) iff $\forall p \in {}^\bullet t$, $W(p, t) \leq m(p)$. If a transition t is enabled by a marking m, then its firing leads to a new marking m' (denoted by $m \xrightarrow{t} m'$) s.t. $\forall p \in P : m'(p) = m(p) + C(p, t)$. The set of markings reachable from a marking m in N is denoted by $R(N, m)$. A run of marked petri net $\langle N, m_0\rangle$ is a path $\pi = m_0 \xrightarrow{t_1} m_1 \ldots \xrightarrow{t_n} m_n$. S. The set of markings reachable from a marking m in N is denoted by $R(N, m)$. The set of markings reachable from a marking m, by firing transitions of a subset T' only is denoted by $Sat(m, T')$. By extension, given a set of markings S and a set of transitions T', $Sat(S, T') = \bigcup_{m \in S} Sat(m, T')$. For a marking m, $m \not\rightarrow$ denotes that m is a dead marking, i.e., $Enable(\{m\}) = \emptyset$.

Two Petri nets $N_1 = \langle P_1, T_1, F_1, W_1\rangle$ and $N_2 = \langle P_2, T_2, F_2, W_2\rangle$, sharing a subset of transitions (resp. places), can be composed by merging these transitions (resp. places) leading to a Petri net regrouping the local attributes of N_1 and N_2. In the approach presented in this paper, we manage to compose Petri nets with disjoint sets of transitions (i.e., sharing only some places).

Definition 2. *Let* $N_1 = \langle P_1, T_1, F_1, W_1 \rangle$ *and* $N_2 = \langle P_2, T_2, F_2, W_2 \rangle$ *be two Petri nets such that* $P_1 \cap P_2 \neq \emptyset$ *and* $T_1 \cap T_2 = \emptyset$. *The composition of* N_1 *and* N_2 *by merging of common places, denoted by* $N_1 \oplus N_2$, *is a Petri net* $\langle P, T, F, W \rangle$ *where* $P = P_1 \cup P_2$, $T = T_1 \cup T_2$, $F = F_1 \cup F_2$ *and* $W : F_1 \cup F_2 \rightarrow \mathbb{N}^+$ *is the mapping assigning the weight* $W_i(f)$ *to any arc in* F_i, *for* $i \in \{1, 2\}$.

The decomposition of a given Petri net into two (ore more) subnets corresponds to the dual operation. Again, the two main decomposition approaches are based on the splitting of a Petri net into two (ore more) subnets that share a subset of places or/and a subset of transitions. The sharing of transitions represent a synchronisation (rendez-vous) between two components while the sharing of places (buffers) represent an asynchronous communication between components.

3.2.1.2 WF-nets

We use a particular Petri net for modeling the control-flow dimension of a service-based processes. This is a variant of Work-Flow nets (WF-nets) that have been introduced in [63].

Definition 3. (WF-net) *Let* $N = \langle P, T, F, W \rangle$ *be a Petri net.* N *is said to be a workflow net (Wf-net) if*

- *there is a set of source place* $I \in P$ *s.t.* ${}^\bullet I = \emptyset$ *and a set of sink places* $O \in P$ *s.t.* $O^\bullet = \emptyset$, *and*
- *for any node* $x \in P \cup T$, *for any source place* $i \in I$ *and for any sink place* $o \in O$, *there exists a path from* i *to* o *which passes through* x.

It should be noted that the sole difference between the WF-nets introduced in [63] and the above definition is the fact that, we allow a WF-net to have several source places and/or several sink places. This has no effect on the semantics of the obtained model but is more convenient for our decomposition approach. As a special kind of Petri nets, WF-nets have the same semantics described above. Its behavior can then be represented by its reachability graph. As far as the behavior of a workflow is concerned, the corresponding WF-net is associated to an initial marking where only the source places are marked with a single token. Besides, a *final marking* of WF-net is every place, that is reachable from a such initial marking, where each sink place is marked and none of the other places is.

3.2.2 Slicing of a Petri net corresponding to a business process

The decomposition of a WF-net corresponding to a business process is based on an on-the-fly traversal of the whole Petri net, considered as a graph, and takes into account the following considerations:

1. each sub-net is a WF-net (according to Definition 3.).

2. the decision to cut a current subnet and start the construction of a new one is based on the structure of the Petri net and on the nature of the business process activities (respectively):

   - when the current node (place) has several output transitions i.e., it is a choice between two (or more) services/activities, (e.g. this may corresponds to an *IF...Else* structure in a BPEL process, to an *OR* operator in a BPMN process),
   - when the current node (place) has several input transitions i.e. it is a conjonction of several exclusive services,
   - when the current node (transition) has several output places i.e., it is the launch of two (or more) parallel services/activities, (e.g. this may corresponds to a *Flow* structure in a BPEL process, to the execution of a component in a BPMN process),
   - when the current node (transition) has several input places i.e., it is a synchronization of several concurrent services,
   - when the current node (transition) is the waiting point for the reception of some asynchronous message (e.g. this may corresponds to the *Receive* activity in BPEL, to the execution of a component implementing receiving messages/parameters in a BPMN process),
   - when the current node (transition) is the synchronization point from some synchronous message, (e.g. this may corresponds to the end of the *Flow* structure in BPEL, to the end of execution of a component in a BPMN process),

3. given the resulting set of subnets, a dependency function, computed on-the-fly, allows to deduce the order of the execution of the services corresponding to these subnets. Thus, the combination of the set of subnets with such a function determines the semantics of the whole net.

4. it is possible to compose back the obtained subnets, by merging shared places, to obtain the original whole WF-net. This allows to preserve the original behaviour (semantics) of the Petri net and hence the corresponding business process.

Given a process model, which has been translated to WF-net, we use Algorithm 1 to slice the WF-net into several WF-subnets. The inputs of the algorithm are a current node ($curNode$), which can be either a place or a transition, a current subnet ($curServ$), to be decomposed, and the whole WF-net (W). The first call to Algorithm 1 is performed using the following input: the source place of the whole WF-net and a current WF-subnet which is empty. Algorithm 1 is recursive and uses the following functions: $NewService()$ creates a new subnet and initializes its set of places, transitions and edges with the given parameters.

Algorithm 1 Decomposition of a WF-net corresponding to a business process.

Require: $initNode, curServ, W$
Ensure: a set of dependent WF-nets services

1: CutNode=$\{receive, synchronous\ invoke\}$
2: $curNode = initNode$
3: **while** ($| curNode^\bullet | = | {}^\bullet curNode | = 1$) & ($curNode \notin CutNode$) **do**
4: add $\{curNode\}$ to curServ
5: curNode=$curNode^\bullet$
6: **end while**
7: **if** (curNode is a place) **then**
8: add $\{curNode\}$ to curServ
9: **end if**
10: **if** Alreadytreated(curNode) **then**
11: **for all** $S \in BuiltSlices(curNode)$ **do**
12: $Dep(curServ, S) =\rightarrow$
13: add S to $BuiltSlices(initNode)$
14: **end for**
15: **end if**
16: **return**
17: $Alreadytreated(initNode) = true$
18: **if** (curNode is a place) **then**
19: **for all** ($t \in curNode^\bullet$) **do**
20: **if** NOT Alreadytreated(t) **then**
21: $S = NewService(curNode, t)$
22: Decomposition(t,S,W)
23: **end if**
24: **end for**
25: **for all** ($t \in curNode^\bullet$) **do**
26: **for all** $S \in BuiltSlices(t)$ **do**
27: $Dep(curServ, S) =\rightarrow$
28: add S to $BuiltSlices(initNode)$
29: **for all** ($t' \in curNode^\bullet \wedge t' \neq t$) **do**
30: **for all** $S' \in BuiltSlices(t')$ **do**
31: $Dep(S, S') = \vee$)
32: **end for**
33: **end for**
34: **end for**
35: **end for**
36: **end if**
37: **if** (curNode is a transition) **then**
38: $S = NewService(curNode, {}^\bullet curNode, curNode^\bullet)$
39: $Dep(curServ, S) =\rightarrow$
40: add S to $BuiltSlices(initNode)$
41: **if** ($| curNode^\bullet | > 1$) **then**
42: **for all** ($p \in curNode^\bullet$) **do**
43: **if** NOT Alreadytreated(p) **then**
44: $S' = NewService(p)$
45: Decomposition(p,S',W)
46: **end if**
47: **end for**
48: **for all** ($p \in curNode^\bullet$) **do**
49: **for all** $S'' \in BuiltSlices(p)$ **do**
50: $Dep(S, S'') =\rightarrow$
51: add S'' to $BuiltSlices(initNode)$
52: **for all** ($p' \in curNode^\bullet \wedge p' \neq p$) **do**
53: **for all** $S''' \in BuiltSlices(p')$ **do**
54: $Dep(S''', S'') =||$
55: **end for**
56: **end for**
57: **end for**
58: **end for**
59: **else**
60: **if** NOT Alreadytreated($curNode^\bullet$) **then**
61: Decomposition($curNode^\bullet$,S,W)
62: **end if**
63: **for all** $S' \in BuiltSlices(curNode^\bullet)$ **do**
64: add S' to $BuiltSlices(initNode)$
65: $Dep(curServ, S') =\rightarrow$
66: **end for**
67: **end if**
68: **end if**

The function $Dep(S, S')$ determines the occurrence dependency between the subnets (services) S and S'. Such a dependency can be of three kinds:

1. $S \rightarrow S'$ means that once the execution of S is finished, the execution of S' should start,

2. $S \vee S'$ means that either S or S' is executed (exclusive or),

3. $S \parallel S'$ means that the execution of S and the execution of S' are concurrent (parallel).

We also save the subnets generated by each node ($curNode$) in $BuiltSlice()$, so that the generation is done once for each encountered node (place or transition) using the boolean function $Alreadytreated()$.

The slicing algorithm is composed of three main phases: First, starting from the current node, the current subnet service ($curServ$) is incremented (by adding places and transitions) as long as we are following a linear (sequential) branch of the Petri net i.e. each encountered node has a single input, a single output, and is neither a receive transition nor a synchronous invoke activity (lines $3-6$). When the first phase terminates, i.e. the current node has more than one input/output node. First, if it is a place then it is the output place of the current subnet (lines $7-9$). Then, if this node has been already treated (lines $10-15$), i.e. a decomposition is starting from this node or a predecessor node has already been performed, then, the set of built subnets must be executed before the current one, and are added as resulting from the decomposition of the initial node ($initNode$). Finally, two cases are considered depending on whether the current node is a place (lines $18-36$) or a transition (lines $37-68$). Notice that, in the second case, we do not distinguish whether the current transition is a "normal" one or a receive/synchronous invoke transition.

Given the set of WF-subnets constructed by our algorithm and the computed dependencies between these subnets, one can orchestrate the execution of the whole process by executing these sub-processes separately. The main novelty in our approach is that such scheduling is not centralized. It is distributed on the different subnets in such a way that the currently executed sub-process is able to compute (using the dependency function) the set of sub-processes that must be enabled after it finishes its execution. Algorithm 2 accomplishes such a task. It has as inputs the current subnet to be executed, a set of subnets SN, and a dependency function Dep. Thus, an execution of the whole WF-net can be obtained by a call to Algorithm 2 with an empty subnet, the set of subnets and the dependency function issued from Algorithm 1. In this algorithm, $Choice$ denotes any **maximal** subset of subnets, denoted by Ch, where any couple $(sn_1, sn_2) \in Ch \times Ch$, $Dep(sn_1, sn_2) = \vee$. Moreover, Dep_S, where Dep is the dependency function and S is a subset of subnets, denotes the projection of Dep on the subset S. Finally, $Dep \setminus Dep_S$ denotes the dependency function obtained by eliminating Dep_S from Dep.

Algorithm 2 starts by waiting while the precondition of the current subnet is not satisfied (lines $1-3$), i.e. the source places are not all marked. Then, the current subnet can be executed (leading to a sequence of firing sequence of transitions) (line 4) before determining the set of subnets to be enabled at the next step $Init$ (line 5). $Init$ contains any subnet sn_i such that there is no other subnet s_j $(j \neq i)$ such that $Dep(sn_j, sn_i) = \rightarrow$. If $Init$ contains more than one element, it can be written as

Algorithm 2 Execution of the current subnet and determination of the next subnets.

Require: *Subnet CurSN*, *Set of subnets SN*, *Dependency function Dep*
Ensure: Executing CurSN and enabling next subnets

1: **while** Precondition of $CurSN$ is not satisfied **do**
2: **wait**;
3: **end while**
4: $run(CurSN)$
5: Set of subnets $Init = \{sn_i \mid \nexists\, sn_j \ : \ Dep(sn_j, sn_i) = \rightarrow\}$
6: **if** $\mid Init \mid > 1$ **then**
7: **for all** *Choice* $ch \subseteq Init$ **do**
8: Let sn_c be some chosen element of Ch (e.g., satisfying a given condition)
9: $Init = (Init \setminus Ch) \cup \{sn_c\}$
10: $Dep = Dep \setminus Dep_{Ch}$
11: $SN = SN \setminus Ch$
12: **end for**
13: **end if**
14: $Dep = Dep \setminus Dep_{Init}$
15: $SN = SN \setminus Init$
16: **for all** $sn_i \in Init$ **do**
17: $Execute - subnet(sn_i, SN, Dep)$
18: **end for**

the following: $Init = ch_1 \cup \ldots ch_k \cup Par$ where ch_i, for $i = 1 \ldots k$ are maximal sets containing exclusive subnets, Par contains concurrent (parallel) subnets (note that, in the first call, Par is necessarily empty), and $ch_i \cap ch_j = \emptyset$ and $ch\ss \cap Par = \emptyset$, for any $i \neq j$. Lines $6 - 15$ allow to keep in $Init$ the subset Par and one representative of each subset ch_i and to update the set of subnets SN and the dependency function Dep. Once we have in $Init$ the set of the subnets to be concurrently enabled at the next step, each one will be enabled and receives the updated set of subnets and the dependency function (lines $16 - 18$). Enabling a subnet is putting a token in the source place which coincides with the sink place of the current subnet. Thus, if the next subnet represents a synchronization between several subnets, it must wait until all these subnets finish their execution i.e., each source place is marked (the precondition becomes then satisfied).

To conclude, Algorithm 2 is associated to each subnet resulting from the decomposition of the whole WF-net and allows each subnet, executed separately, to autonomously launch the subnets to be executed in the next step. The first call is performed with an empty subnet unless the first enabled subnet is determined, a priori, by executing lines $5 - 15$ in which case the first call can be performed by the subnet resulting from this execution.

3.2.3 Proof of preservation of semantics

Theorem 1 provides the execution chain to follow for obtained subnets to retrieve the business functionality of the initial business process.

Theorem 1. *Let SN and Dep be respectively the set of subnets and the dependency function resulting from the application of Algorithm1 on a WF-net N. Let σ be a sequence of transitions. σ is a firing sequence of N (i.e. $\sigma \in L(N)$) iff σ corresponds to an execution of Algorithm 2 with an empty subnet.*

Proof.

- $\Rightarrow$

  Let σ be a run of SN and let us demonstrate, by induction on the length of σ ($|\sigma|$), that σ can be generated by Algorithm 2.

  - $|\sigma| = 1$. Assume $\sigma = t$ and let i the source place of SN. Then, $t \in i^\bullet$ and $m_0(i) > Pre(t, i)$ (where m_0 is the initial marking). Thus, the precondition of t is satisfied and the run of some of the subnets containing i (line 4) will allow to fire the transition t. Note that it is possible that i belongs to several subnets when, in SN, $|i^\bullet| > 1$.
  - Assume that any run of SN, of length $n \in \mathbb{N}$, can be generated by Algorithm 2.

    Let $\sigma = t_1...t_{n+1}$ be a run of SN. Then, the sequence $t_1...t_n$ can be generated by Algorithm 2. We distinguish the two following cases:

    1. t_n and t_{n+1} belong to the same subnet sn_i . In this case, the run of sn_i , by the instruction at line 4 allows to fire t_{n+1} directtly after the firing of t_n.
    2. t_n and t_{n+1} belong to two different subnet sn_i and sn_j respectively. In this case, only one of the following conditions holds:
       (a) $Dep(sn_i, sn_j) = \rightarrow$
       (b) $Dep(sn_j, sn_i) = \rightarrow$
       (c) $Dep(sn_i, sn_j) = ||$

       Indeed, the value of $Dep(sn_i, sn_j)$ can not be $\vee$ otherwise t_{n+1} would not be friable, within SN, after the firing of t_n.

       * The two first cases being symmetrical, assume that $Dep(sn_i, sn_j) = \rightarrow$ Then, there exists a set of subnets Par_i such that $\forall sn_k \in Par_i$, $\forall sn_l \in Par_i$, $Dep(sn_k, sn_l) = || \wedge Dep(sn_k, sn_j) = \rightarrow$. If such a set is empty let $Par_i = sn_i$. There exists $m < n$ s.t. for any subnet

$sn_k \in Par_i$ there exists a run σ_k of sn_k s.t. $\sigma^m_{k_{sn_k}} = \sigma_k$ (where σ_m is the suffix of σ starting at the transition t_m and $\sigma^m_{k_{sn_k}}$ denotes the projection of this run on the transitions of sn_k). Indeed if this does not hold, then t_{n+1} would not be enabled within SN. Thus, the precondition of sn_j becomes satisfied (line 1) as soon as t_n is fired and t_{n+1} is fired by the run of sn_j (line 4).

- Assume now that $Dep(sn_i, sn_j) = ||$. Then, sn_i and sn_j can be launched in parallel by the loop instruction at line 16. One can then choose an interleaving allowing to fire t_{n+1} directly after t_n.

- $\Leftarrow$

  Let $\sigma = t_1...t_n$ be a sequence of transitions (execution) generated by Algorithm 2 and let us assume that σ is not a firing sequence of SN. Note first that t_1 is fireable at the initial marking of SN, otherwise the precondition (line 1) would not be satisfied and t_1 would not start any sequence generated by Algorithm 1. Thus, if σ is not a firing sequence of SN, then there exists $2 \leq i \leq n$ such that $t_1...t_{i-1}$ is a run of SN and t_i is not fireable by the marking reached by this run. We distinguish the two following cases:

  1. t_{i-1} and t_i belong both to a subnet sn_i . In this case, t_i is not enabled by SN is not possible since instruction 4 launch the subnet sn_i which has the control on all its transitions i.e. the firing of t_i depends only on the firing of t_{i-1}.
  2. t_{i-1} and t_i belong to sn_k and sn_l $(k = l)$ respectively. t_i is not friable means that there exists a place $p_i \in {}^\bullet t_i$ whose marking, after the firing of t_{i-1}, is strictly less than $Pre(t_i, p_i)$. This is not possible because the fact that t_i is generated by the algorithm (line 4) means that the precondition of the subnet sn_l is satisfied i.e., all the input places of t_i have been sufficiently marked by the execution of the loop at line 16.

3.2.4 Example: Slicing of the Online shop process

For this example, we recall the online shop BPEL process introduced in Section 2.1.3. To perform the BPEL to Petri net transformation, we use the *BPEL2PN*[1] tool. *BPEL2PN* tool is a Java-based compiler that transforms a process specified in BPEL into a Petri net according to the Petri net semantics [64]. The output format of *BPEL2PN* is a Petri net in the data format of the Petri net based model checker LoLA [65] [66]. LoLA also offers the opportunity to write out the net into the standard interchange format for Petri nets, the Petri Net Markup Language (PNML) [67]. Thus, since other modeling languages, which are more frequently used in practice, map to Petri nets (for BPEL, see e.g. [68], for BPMN, see e.g. [69]), our approach is relevant for

[1] http://www2.informatik.hu-berlin.de/top/bpel2pn/index.html

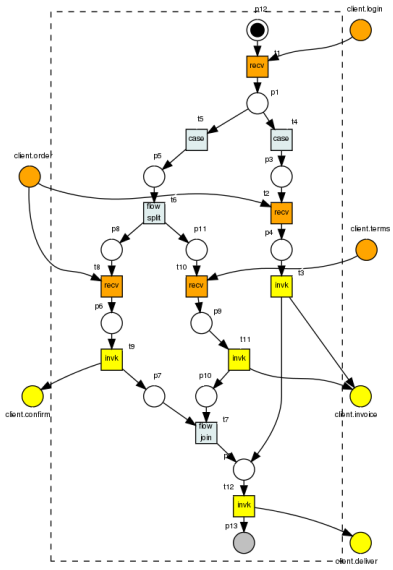


Figure 3.1: The Petri Net corresponding to the shop process.

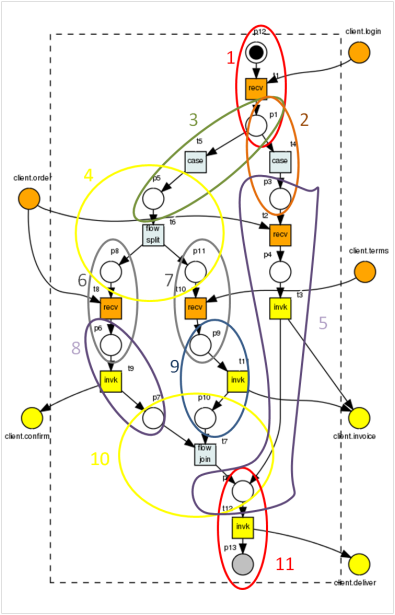


Figure 3.2: The decomposed Petri Net corresponding to the shop process.

a very broad class of modeling languages. The generated Petri net from our shop process is schematized in Figure 3.1. It should be noted that in all our examples we used BPEL processes without handlers when we used the *BPEL2PN* tool.

The execution of Algorithm 1 allow slicing the Petri net into a set of dependent WF-nets (See Figure 3.2) and calculation of the dependency function *Dep* (See Table 3.1) [70]. After that, we generate and aggregate the java services code corresponding to each subnet using our *BPEL2Java* tool. This program generates java services from BPEL activities based on the transformation rules that we have defined. More details about the *BPEL2Java* tool and defined transformation rules are provided in Section 6.2.1.

3.3 Slicing of applications based on services compositions

To slice applications based on services compositions, we opted for the use of directed graph to formally respresent the application components before performing the slicing. The use of a graph-based composition enables us to focus on all application components interactions and dependencies between them when processing the slicing.

Preliminaries on graphs and directed graphs are introduced in Section 3.3.1. The nodes of directed graphs represent the applications services (e.g. the application components in the case of an SCA-based application), the edges represent compositions between these services (e.g. component bindings in the case of an SCA application)

	sn_1	sn_2	sn_3	sn_4	sn_5	sn_6	sn_7	sn_8	sn_9	sn_{10}	sn_{11}
sn_1	-	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
sn_2	-	-	$\vee$	$\vee$	$\rightarrow$	$\vee$	$\vee$	$\vee$	$\vee$	$\vee$	$\rightarrow$
sn_3	-	-	-	$\rightarrow$	$\vee$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
sn_4	-	-	-	-	$\vee$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$	$\rightarrow$
sn_5	-	-	-	-	-	$\vee$	$\vee$	$\vee$	$\vee$	$\vee$	$\rightarrow$
sn_6	-	-	-	-	-	-	$\|\|$	$\rightarrow$	$\|\|$	$\rightarrow$	$\rightarrow$
sn_7	-	-	-	-	-	-	-	$\|\|$	$\rightarrow$	$\rightarrow$	$\rightarrow$
sn_8	-	-	-	-	-	-	-	-	$\|\|$	$\rightarrow$	$\rightarrow$
sn_9	-	-	-	-	-	-	-	-	-	$\rightarrow$	$\rightarrow$
sn_{10}	-	-	-	-	-	-	-	-	-	-	$\rightarrow$

Table 3.1: Dependency function of the decomposition of the shop process

and the edge directions indicate the call ways. The slicing algorithm that we have addressed is presented in Section 3.3.2. This algorithm slices a given application into a set of elementary and autonomous services based on its directed graph representation and information in its descriptor (e.g. a *.composite* file in the case of an SCA-based application). The services compositions are replaced by remote calls. An illustrative example of an SCA-based application slicing is provided in Section 3.3.4.

3.3.1 Preliminaries: graphs, directed graphs

In the following, we introduce both graph and directed graph notions.

Definition 4. (Graphs & directed graphs) *A graph $G = \langle V, E \rangle$ comprises a set $V = \{1, ..., n\}$ of n nodes, and a set E of directed edges where $(i, j) \in E$ is an edge from node i to node j. We further associate with each edge (i, j) a number δ_{ij} that expresses the relative hierarchy of the nodes. Usually, $\delta_{ij} = 1$ for a directed edge $i \rightarrow j$, meaning that i precedes j by one unit.*

3.3.2 Slicing of a directed graph corresponding to an application based on services compositions

Given an application based on services compositions which have been translated to a directed graph, we address Algorithm 3 to slice it into several aggregated services. The input of the algorithm is the directed graph G characterized by set of nodes (V) and edges (E). The algorithm provides as result a *serviceList* structure. Each element of *serviceList* describes an application component and all its sub-elements (e.g. service, references, binding, properties, etc.). This algorithm is executed only once to slice an application before its deployment.

Algorithm 3 Slicing and aggregation of an application based on services compositions.

Require: $G = \langle V, E \rangle$
Ensure: *serviceList*
1: $serviceList = \emptyset$
2: n= $sizeOf(V)$
3: k=1
4: **while** ($k <> n$) **do**
5: $serviceList(k) \leftarrow V_k$
6: $AggregateService(serviceList(k))$
7: **for all** $t \in E$ **do**
8: **if** (δ_{kt}) **then**
9: $GenerateClientInterface(V_k, V_t)$
10: **end if**
11: **end for**
12: k=k+1
13: **end while**

Algorithm 3 traverses the graph and assigns each node to a *serviceList* cell (line 5) before performing a call to *AggregateService* function (line 6).

$AggregateService(serviceList(k))$ process service corresponding to V_k code generation and its validity regarding standard service specifications. The service code is copied from the component source code. In the case of an SCA-based application, the SCA annotations in the components source code are parsed and transformed to a regular code. For example, a *@Property* annotation, which provides the name of a given property of a component, is transformed in the service code to a simple or a complex types. A *@Reference* annotation, which allows a given SCA component implementation to call another component, is transformed to a local call and the *@Remotable* annotations are transformed to remote calls.

After service aggregation, the algorithm checks if the selected graph node have outgoing edges (line 7) to other nodes (i.e. (δ_{kt}) =1, $\forall t \in E$). This indicates if the service corresponding to this node has interactions with other services of the application. For all these services, a call to $GenerateClientInterface$ is made (lines $8 - 10$).

$GenerateClientInterface(V_k, V_t)$ adds a client code to the service code corresponding to V_k to allow it to invoke service corresponding to V_t. This remote call is carried through configuration of generic clients. The needed properties (i.e. name of the target service, name of the operation to invoke, input parameters number and types, etc.) to set up the clients are determined from the application descriptor (e.g. the *.composite* file in the case of an SCA-based application).

3.3.3 Proof of preservation of semantics

The intermediate formal representation of application based on services compositions in directed graphs before slicing is strictly faithful to the structure of the initial application. Furhermore, we reuse the same components' bindings specified in the application descriptor to perform the interactions and ensure the communications between the sliced services. Finally, we use equivalent sources when aggregate the services.

3.3.4 Example: Slicing of the ComputePrice application

For this illustrative example, we recall the ComputePrice application introduced in Section 2.1.4. The representation of this application in a directed graph in schematized in Figure 3.3.

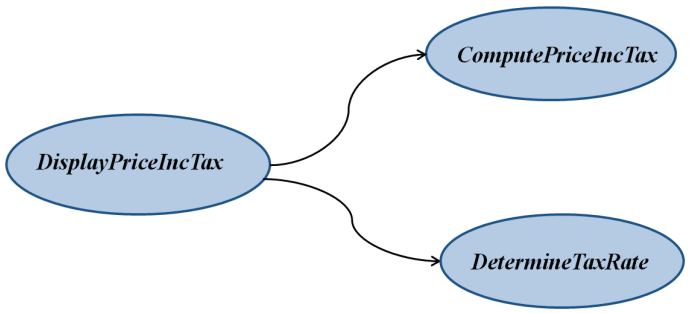


Figure 3.3: The directed graph respresenting the ComputePrice application.

The execution of Algorithm 3 ensures:

1. Parsing the ComputePrice application *.composite* descriptor,
2. Initializing the *serviceList* structure with the three components,
3. Processing the slicing and copy components' code to generated services' code,
4. Aggregating generated services code.

For action 3 and action 4, since the *DetermineTaxRate* component is a remote Java Web service, we process only slicing, code generation and service aggregation of *DisplayPriceIncTax* and *DisplayPriceIncTax* components. The aggregation actions involve the *DisplayPriceIncTax* and *DisplayPriceIncTax* components annotations processing and client code integration (See Figure 3.4).

More details about Algorithm 3 implementation and defined transformation rules are provided in Section 6.2.1.

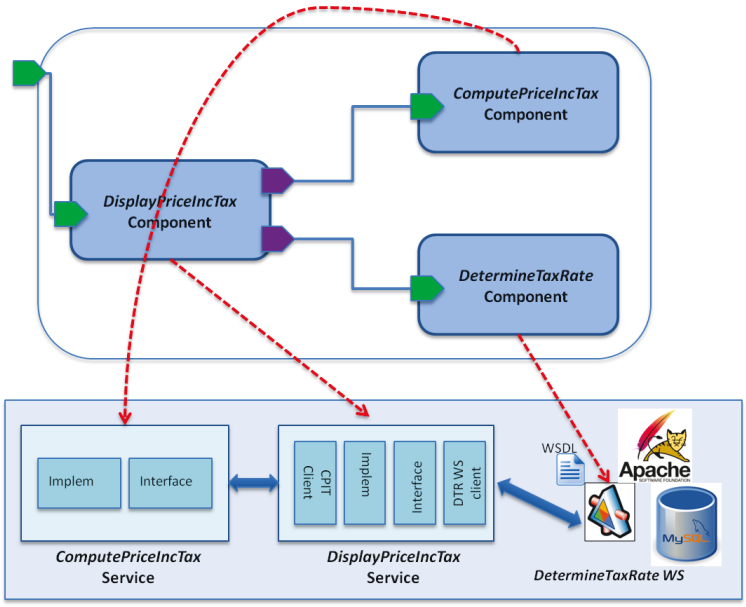


Figure 3.4: Obtained services after ComputePrice application slicing and aggregation.

3.4 Conclusion

In this Chapter, we presented and detailed the first step of our defined SPD approach i.e. slicing service-based applications into a set of elementary services. In this step, we considered: (i) applications modeled as business processes and can be formally represented using Petri nets and (ii) applications modeled as compositions of service components and can be represented using graph-based composition. The use of formal representations allow us to preserve semantics of sliced applications.

For applications modeled as business processes, we defined algorithms to slice their correspondent Petri net into a set of dependent WF-nets and to determine the orchestration to follow for their execution. We also provided the proof of preservation of initial business process semantics when executing the WF-nets. An illustrative example of a BPEL process slicing was given.

For applications modeled as compositions of service components, we defined an algorithm to slice its correspondent directed graph into a set of elementary services and aggregate them. Aggregation tasks aim at to copy and adapt the services code from the application code. Client stubs are also added to services to make them able to interact. An illustrative example of an SCA-based application slicing was provided.

In the next Chapter, we detail the second step of the SPD approach i.e. package obtained services in appropriate service micro-containers.

Chapter 4

Service Packaging

Contents

4.1 Introduction

In this Chapter, we detail the second step of the SPD approach. The first step of our defined approach involved slicing of a service-based application into a set of elementary and autonomous services (See Chapter3). This step aims at packaging the obtained services in micro-containers before deploying them in a target Cloud environment. The packaging step consists in generating an appropriate micro-container around one service with the minimal modules implementing its required resources (e.g. specific communication bindings). This step is performed when the target Cloud environment do not support provisioning required Cloud resources for hosting and execution of sliced services. The choice of the micro-containers was motivated by (1) the possibility to provision dynamically services' required resources independently of the target Cloud capacities and (2) its higher performances against classical service containers (e.g. Apache Axis) demonstrated in Section 6.3.

Service micro-containers provide the minimal functionalities to manage hosted service life cycle according to the definition introduced in [71]. These basic functionalities

ensure the minimal main process of our micro-containers (e.g. services hosting, interaction with clients, etc.). For example, we failed to incorporate a safety module for managing access since it is a prototype and a service balancer module as we are assuming a single service per container. However, the design of the micro-containers was made so that these modules can be added as extensions or add-ons if necessary. For example, we add extensions to support services non-functional properties (e.g. migration) if they are requested by the developer. These extensions can be integrated to the generated micro-containers to provide specific features at a very fine degree of granularity (i.e. service level). For example, when adding migration facilities to a given micro-container, we can migrate its service from a hosting machine to another. This prevents us to migrate the entire machine with all its other running services.

We thought of designing a system composed of two main parts:

1. The service micro-container,
2. The generic packaging platform that build the micro-container and package the service to host in it.

This Chapter is organized as follows: We present our performed packaging framework and comment the packaging process of a given service in a micro-container in Section 4.2. Then, we detail the architecture of the generated micro-containers in Section 4.3. The service micro-container might include one or more non-functional properties if they are required by the developer such as migration, monitoring and/or elasticity. These extensions are detailed respectively in Sections 4.4, 4.6 and 4.5. Finally, illustrative examples consisting in packaging processes of services obtained from slicing the online shop process (See Section 3.2.4) and ComputePrice application (See Section 3.3.4) are detailed in Section 4.7.

4.2 Service packaging framework

Since we consider several types of services (languages, bindings, etc.), we are able to dynamically generate the correspondent micro-container from the packaging framework for each service to be deployed [58] [59] [72]. An overview of the main components of the packaging framework and architecture of generated micro-containers are detailed in Figure 4.1.

To package a service and build its appropriate micro-container, one must mainly provide for the deployment framework two elements:

1. The service to package with all its components (code, resources, etc.),
2. A deployment descriptor that specifies the container options.

The *Processor module* analyzes the service code, parses its associated descriptor to determine the service binding types and instantiates an appropriate *Communication*

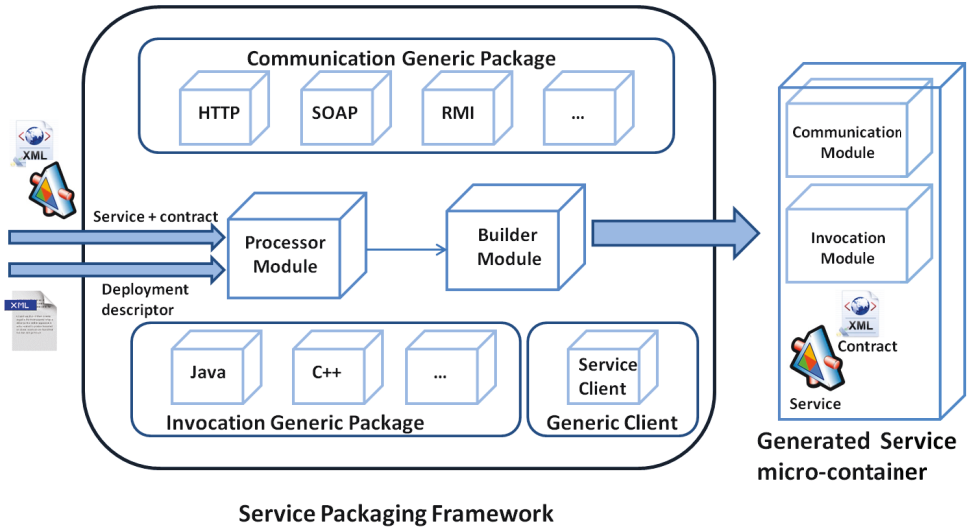


Figure 4.1: Service micro-container packaging framework.

module implementing these bindings from the *Communication Generic Package*. The instantiated *Communication module* is associated to a generic service container sources by the *Builder module*. The same principle is also followed for the selection of the *invocation module* to run the service once packaged in the micro-container. Based on the service implementation programming language, the appropriate *invocation module* is instantiated from the *Invocation Generic Package*.

The resulting code represents the generated micro-container code. It is composed only of the necessary modules for the deployed service, no more, no less. The generated micro-container hosts the service and implements its bindings regardless its communication protocol support as long as they are included in the *Generic Communication Package* and regardless the programming language as long as they are included in the *Invocation Generic Package*.

4.3 Service micro-container

Each generated service micro-container consists at least of three modules (See Figure 4.1):

- *Communication module* to establish communication and to support connection protocols,
- *Invocation module* to process ingoing and outgoing data into and out of the server (packing and unpacking data),
- *Service module* to store and invoke the packaged service and its contract (service descriptor).

The packaging framework provides also a generic client to invoke packaged service in the generated micro-container. The client setup is based on the service bindings type and information described in its contract. For example, for a Java Web service the contract is a WSDL document and needed information to setup the client are described in *operation*, *input*, *output* and *service* elements.

Service micro-containers process client requests according to the following scenario:

1. Receiving of the client request,
2. Extracting communication protocol envelops (e.g. HTTP SOAP header),
3. Invocation of the packaged service,
4. Building of the response message,
5. Send the response back to the client.

4.4 Adding migration facilities to service micro-containers

We extended our micro-containers allowing them to be mobile and give them the ability to migrate, with its service inside, from one host to another once deployed in a Cloud environment [73]. To handle this, we used two different approaches:

- JADE-based technology integration in the packaging framework,
- Extension of the packaging framework by adding a generic migration module.

JADE-based migration uses an existent mobile multi-agent platform called Java Agent DEvelopment Framework [1] to handle service migration. We have integrated this platform to our service packaging process. The choice of JADE was motivated by the analysis presented in [74]. In this survey, the authors dress a comparison between existent mobile agent platforms and show that JADE is the most appealing. JADE is FIPA-compliant so it allows interoperability between agents and provides many graphical tools for development and debugging.

The second approach followed consists in extending the packaging framework by adding a generic migration package which component's can be instantiated and parameterized during the packaging phase. Both approaches are detailed respectively in Section 4.4.1 and Section 4.4.2.

4.4.1 JADE-based migration technology integration

The idea we adopted was to encapsulate a micro-container in a mobile agent as detailed in Figure 4.2.

[1] http://jade.tilab.com/

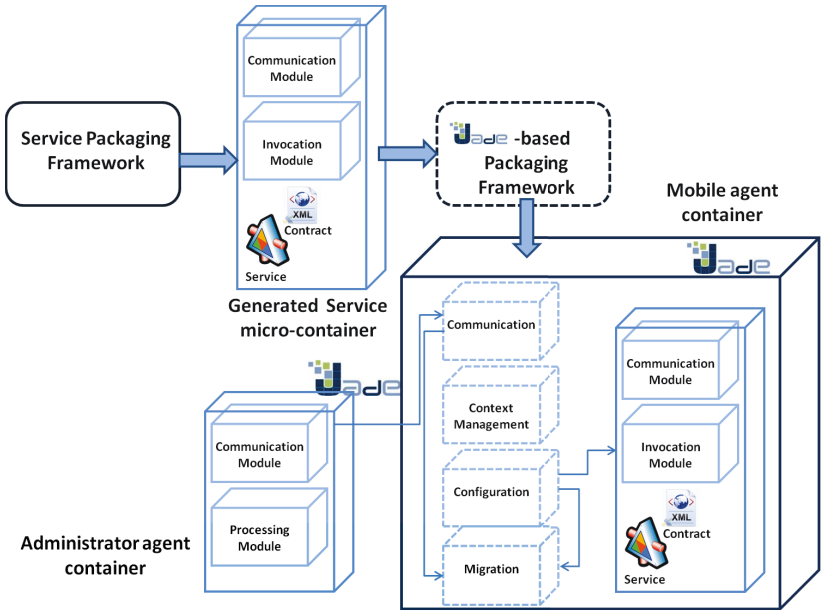


Figure 4.2: JADE-based mobile micro-container system.

We developed a new JADE-based framework that takes as input the generated micro-container from our packaging framework and encapsulates it in a mobile agent container. This new framework is an extension to the procedure for the service packaging process.

Mobile agent architecture is illustrated in Figure 4.2. It includes the generated micro-container and other specific components useful to mobile agent management. Keeping with the philosophy of micro-containers building process, we selected only the minimum execution environment necessary for the mobile agents.

Migration order is given by a specific deployed container called Administrator agent container which determines when and where to move the service. Communication between the Administrator agent container and the mobile agent container are based on Java Remote Method Invocation[2].

This kind of migration is called reactive migration which is different from the proactive one in which the agent itself takes the decision. The mobile agent contains specific JADE technology components.

4.4.2 Adding generic migration package to the packaging framework

Unlike the JADE-based migration approach, this approach consists in intervening on micro-containers sources and adding migration facilities. To do this, we extend our packaging framework by adding a new generic migration package (See Figure 4.3).

[2]http://docs.oracle.com/javase/tutorial/rmi/

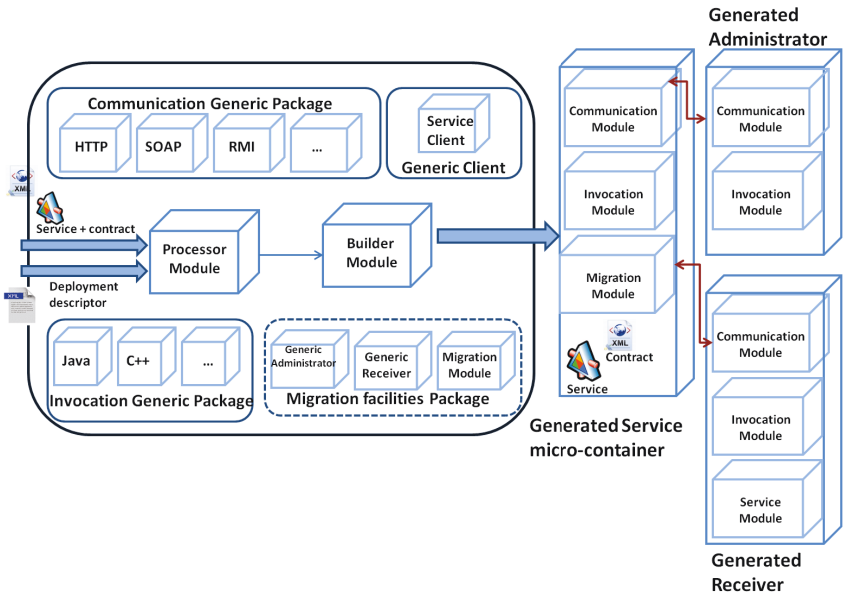


Figure 4.3: Extended packaging framework with generic migration component.

We included a migration module to add to generated mobile micro-containers at service packaging time. This module is instantiated from the *Migration* component and is responsible of interacting with receivers to perform migration to a target host. Migration requests are sent by an administrator container instantiated from the *Generic administrator* component. An administrator container and a receiver can manage a set of mobile micro-containers (e.g. an administrator and a receiver per host for example). Generated receivers are instantiated from *Generic receiver* and are composed of three modules:

- *Communication* module to allow receiving requests from micro-containers to migrate,
- *Invocation* module to process in-going data from the host where it is located,
- *Service* module to download micro-containers code classes and deserializes it.

The scenario of migration is described in Figure 4.4. The administrator sends a migration order indicating the new destination to a mobile micro-container. Then, the micro-container stops receiving new client requests and finishes the processing of received requests. After that, the migration module serializes the micro-container's code and sends a message to the receiver. The receiver address is communicated by the administrator as a parameter. The service module of the receiver downloads then the serialized code and deserialize it. Finally, the micro-container is restarted in the new host and is available for new client requests.

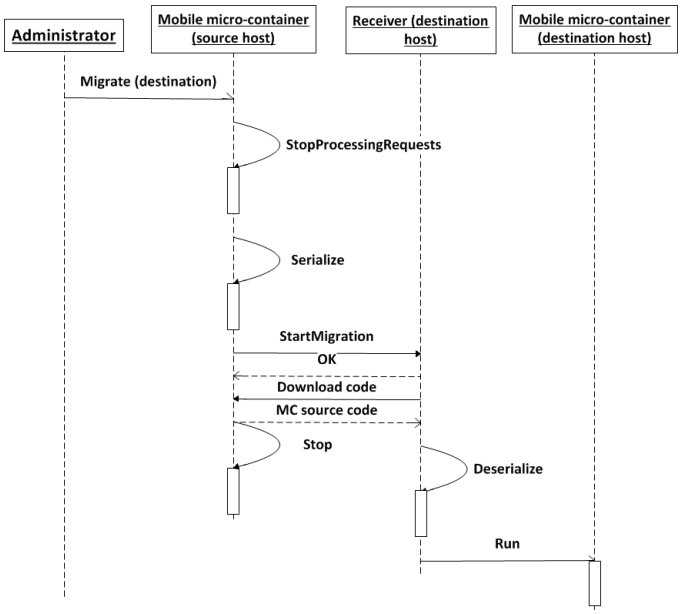


Figure 4.4: Mobile micro-container migration steps.

4.5 Adding elasticity facilities to service micro-containers

In order to add elasticity facilities to micro-containers, we extended the packaging framework with adding new generic modules implementing elasticity mechanisms. The added modules are schematized in Figure 4.5.

We have not made changes in the micro-container itself. However, we have include generation of two additional components in the packaging process to implement and process the target elasticity mechanisms. On one hand, we designed a controller container to monitor a designated set of services packaged in micro-containers. The controller is instantiated from the *Generic controller* component which implements a process controller according to a given formal model [75]. On the other hand, we designed a set of front-end routers (i.e. one per generated service micro-container). A router is instantiated from the *Generic Router* component and is assimilated to a micro-container proxy. The invocation of a service packaged in a micro-container must henceforth be processed via the dedicated router. Indeed, the router is responsible of load balancing between all of micro-container instances and ensures then abstraction of all service copies at duplication/consolidation time. The architecture of the routers are strongly inspired from micro-containers architecture to optimize changes on the packaging framework. The routers does not contain services but still contains their contracts in order to facilitate prospective service invocations. The service copies management is insured thanks to a routage table.

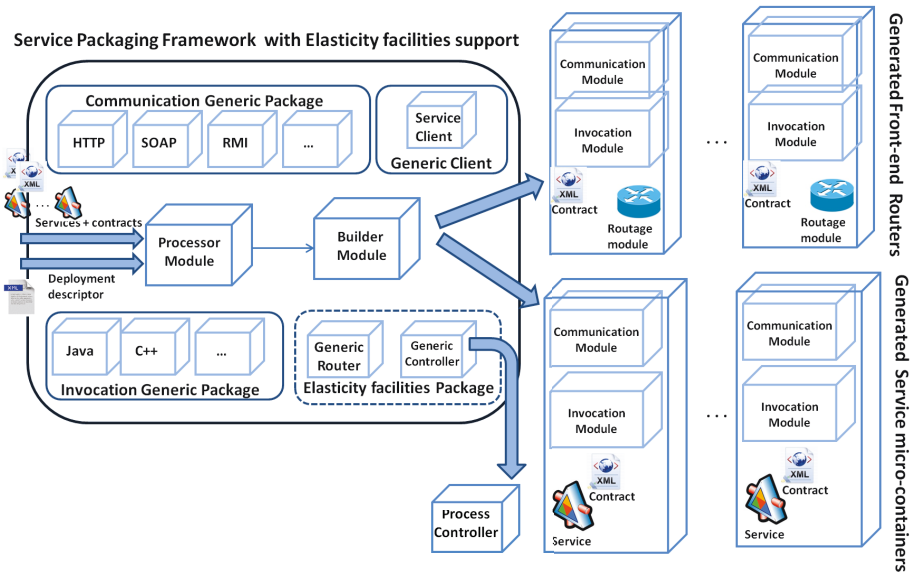


Figure 4.5: Service micro-container packaging framework with elasticity facilities support.

4.6 Adding reconfiguration and monitoring facilities to service micro-containers

A related work has focused on adding reconfiguration and monitoring facilities to service micro-containers [76] [77]. To achieve this, the packaging framework was extended by adding new generic modules supporting these features (See Figure 4.6).

Packaging monitoring and reconfiguration facilities in micro-containers principle is the same than packaging migration facilities. Based on information given in the deployment descriptor, the appropriate module is instantiated and packaged from the *Monitoring facilities package*. Two monitoring models are supported: Monitoring by polling and monitoring by subscription. The first one allows to request the current state of a packaged service whenever there is a need. The second one is based on a publish/subscribe system which is defined as a set of nodes divided into publishers and subscribers. There are two modes of monitoring by subscription: on interval and on change. For the on interval mode, the service state broadcasting is performed by the publisher (i.e. the micro-container) periodically to subscribers (e.g. clients, platform administrator, etc.). The on change mode consists on broadcasting the service state when a specific event occurrs (e.g. method call).

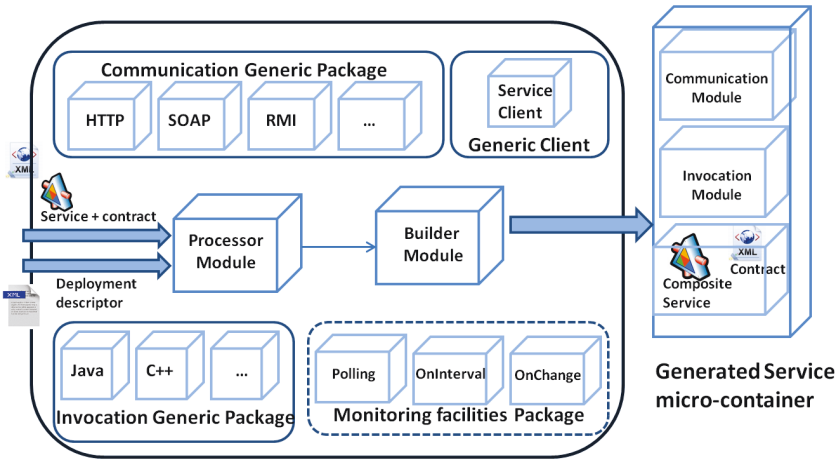


Figure 4.6: Extended packaging framework with generic monitoring and reconfiguration facilities.

4.7 Example: Packaging of Shop process and ComputePrice services

After slicing the shop process and ComputePrice and aggregate obtained services (See respectively Sections 3.2.4 and 3.3.4), we provide these services to our packaging framework in order to package each one of them in an appropriate micro-container according to the process described in Section 4.2. The obtained set of micro-containers are standalone and executable applications (i.e. Java ARchive files) that can be run over any Java Virtual Machine (JRE-based) environment. Once started, the service-micro-containers are listening to prospective requests for hosting services execution. To execute the whole application, we simply send a request to the micro-container hosting the first service. The orchestration between the micro-containers ensures the same semantic functionality of the initial service-based application. Call requests and interactions between service micro-containers are made through a parameterized generic SOAP client (See Section 6.2.1 for more details).

4.8 Conclusion

In this Chapter, we detailed the second step of the SPD approach i.e. packaging application's sliced services in appropriate micro-containers. The service micro-containers are used to package dynamically a sliced service with its required execution resources before deployment time. This is useful when the target Cloud environment do not support provisioning of such resources. Only necessary resources to implement service binding types, such as communication protocols, are selected from the packaging framework and encapsulated in the generated micro-container to host the service. We

also performed packaging framework extensions to add non-functional service properties support in generated micro-containers if needed. At the end of the Chapter, we provided details about packaging obtained services after online shop process and ComputePrice application slicing.

In the next Chapter, we present the third step of the SPD approach i.e. deploying the packaged services in existing Cloud environments.

Chapter 5

Service Deployment

Contents

5.1 Introduction

In this Chapter, we detail the third step of our defined SPD approach. This step is the last one of the approach and covers the deployment in a target Cloud after slicing a service-based application into several elementary services (See Chapter 3) and packaging them in appropriate micro-containers (See Chapter 4) [62].

According to [78], there are four types of service deployment solutions i.e. manual, script-based, language-based and model-based deployment. Since we consider our service micro-containers as standalone and autonomous applications, we opted for the manual deployment.

There are two ways to deploy micro-containers: The first one consists in deploying them as standalone applications in a Cloud infrastructure (IaaS), while the second one consists in deploying them in a Cloud platform (PaaS). Challenges and requirements are different for an IaaS or a PaaS deployment.

Deploying service micro-containers in IaaS consists in uploading and running the micro-containers in virtual machines (VMs) instantiated from an infrastructure manager such as OpenNebula[1] or OpenStack[2]. Meanwhile, the micro-containers can also be deployed in existing Cloud platforms as standalone applications (e.g. See [79] for deployment in Cloud Foundry, See [80] for deployment in Heroku). This deployment is based on the Cloud platform description models and is performed through their user APIs. It should be noted that for an IaaS deployment, unlike for a PaaS deployment, the developer have to install and configure all prospective resources needed by the application apart service micro-containers (e.g. installation of a database, configuration of the container binding with the database, etc.). Such manipulations add complexity to the deployment task and it is contradictory to Cloud operating principles. Indeed, according to the definition of the Cloud Computing paradigm [11] and its correspondent economic model [81] [19], installation and configuration tasks related to the deployment should be insured and delegated to the Cloud environment.

Based on this, we can say that a deployment in a PaaS is more appropriate for our use case. So, as part of our work, we defined a generic description model for applications and PaaS resources to generalize the deployment procedure in Cloud platforms. Our defined model allows seamless interactions with different and heterogeneous PaaS and address applications portability issues (See Section 1.2). Our model extends the OCCI standard and provide unified operations through a REST API that we called COAPS for applications provisioning and management in Cloud platforms.

Once deployed, the micro-containers are started and begin to listen to prospective client requests for invoking packaged services. To execute a deployed application, a client sends an invocation request to the micro-container that hosts the first service of the application. The call sequence between the different micro-containers ensures the semantic functionality of the whole initial service-based application and returns the same execution result.

This Chapter is organized as follows: We present and comment our defined generic description model for platform and application resources in PaaS in Section 5.2. COAPS API is a REST API implementing this model and providing generic interfaces for applications and PaaS resources provisioning. COAPS API specifications are detailed in Section 5.3. Finally, the deployment procedures of the micro-containers packaging online shop and ComputePrice services are described in Section 5.4.

[1] opennebula.org
[2] openstack.org

5.2 Model for PaaS resources description and provisioning

We defined an OCCI-based model for the description of the platform and application resources independently from the targeted PaaS [82]. Our proposed model extends the OCCI core model and is composed of two main parts:

1. An OCCI platform extension which describes all PaaS resources that can be provisioned by a PaaS to set up an appropriate environment,

2. An OCCI application extension which describes the application resources to deploy in this environment.

These two extensions define and classify PaaS resources through generic OCCI types that we have defined. Each one of these types is characterized by a set of attributes and actions to handle and manage it according to the OCCI standard. By doing so, provisioning and management processes of these resources can be aggregated which brings us to handle them on the same way through our defined properties and actions independently of the hosting PaaS. Our two defined extensions are detailed in the rest of the Section.

5.2.1 Platform resources description model

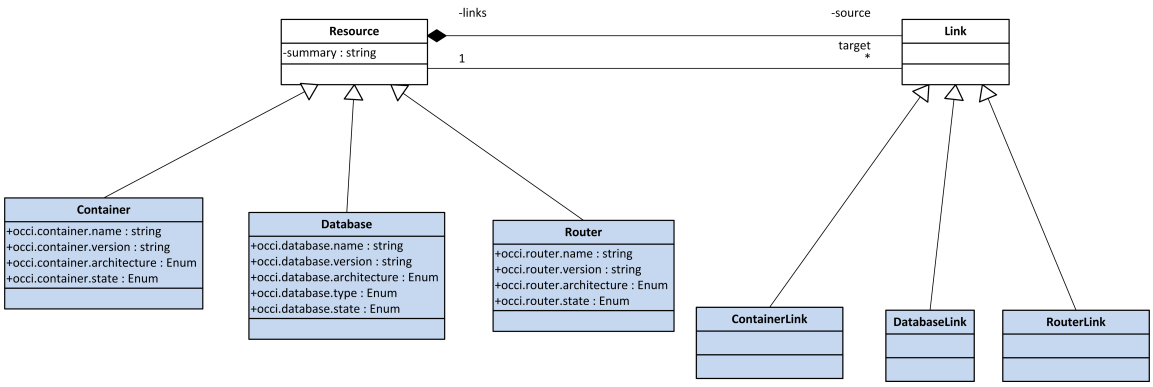


Figure 5.1: Overview of the defined OCCI platform types.

The first part of our extension describes the platform resources layer. These resources allow definition and instantiation of an application hosting-environment. Figure 5.1 illustrates our defined platform types. Platform resource types are derived from *Resource* type at OCCI core level whereas platform resource mixins are derived from the *Mixin* type and the interfaces, which links these resources between them, are derived from the *Link* type.

The main defined OCCI platform resources are:

- *Database* resources which are data store resources for platform applications processing persistent data (e.g. MySQL, PostgreSQL, CouchDB, etc.),
- *Container* resources which are engines to host and run applications (e.g. Apache Axis container, Bonita, IBM WebSphere, etc.),
- *Router* resources which are resources that provide protocols, message format transformations and routing (e.g. ESBPetals router, Apache Synapse, etc.).

We also define a set of links to connect and interact with these resources:

- *ContainerLink* to connect to *Container* resources,
- *RouterLink* to connect to *Router* resources,
- *DatabaseLink* to connect a *Container* resource to a *Database* resource.

The *Kind* instances defined for each one of the platform *Resource* or *Link* subtypes are described in Table 5.1. In addition to that, a set of platform mixin resources can be defined if needed through this extension to support specific platform resource features (e.g authentication PaaS features, logging PaaS features, etc.).

Table 5.1: The kind instances defined for the platform subtypes of Resources and related Links.

Term	Scheme	Title	Related Kind
Database	$< schema > /platform\#$	Database Resource	$< schema > /core\#resource$
Container	$< schema > /platform\#$	Container Resource	$< schema > /core\#resource$
Router	$< schema > /platform\#$	Router Resource	$< schema > /core\#resource$
DatabaseLink	$< schema > /platform\#$	Database Link	$< schema > /core\#link$
ContainerLink	$< schema > /platform\#$	Container Link	$< schema > /core\#link$
RouterLink	$< schema > /platform\#$	Router Link	$< schema > /core\#link$

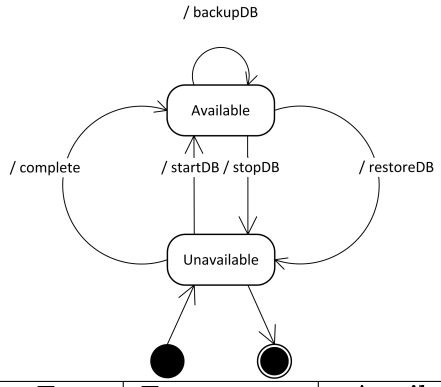
Defined platform resources (i.e. *Database*, *Container* and *Router*) are characterized by a set of attributes (including OCCI default attribute *state*) and actions according to OCCI core model resources. We believe that platform resources, which can be provisioned by a PaaS, may be assigned to one of these three resources.

The Database resource type

The *Database* resource type represents storage resources which can be provisioned by a PaaS provider for applications which process persistent data. *Database* resources can be relational (e.g. MySQL, PostgreSQL, etc.) or non-relational (e.g. MongoDB, CouchDB, etc.). The Database type inherits the *Resource* base type defined in the OCCI core model.

Table 5.2: Database type attributes.

Attribute	Type	Multiplicity	Mutability	Description
occi.database.name	String	0...1	Mutable	Name of the instance.
occi.database.type	Enum {relational, KeyValue, document, graph}	0...1	Mutable	Scheme type of the instance.
occi.database. architecture	Enum {x86, x64}	0...1	Mutable	CPU architecture of the instance.
occi.database.version	String	0...1	Mutable	Version label of the instance.
occi.database.state	Enum {available, unavailable}	1	Immutable	Current state of the instance.



Action Term	Target state	Attributes
StartDB	Available	DB credentials
StopDB	Unavailable	-
RestartDB	Available	DB credentials
BackupDB	None	-

Figure 5.2: State diagram and actions applicable to Database type instances

Table 5.2 details the attributes describing the *Database* type through its *Kind* instance (Database scheme). The *state* attribute indicates the current state of a given instance. The execution of an action induces to the modification of its value according to the diagram presented in Figure. 5.2. For example, for an already created and unavailable *Database* instance, the execution of the *StartDB* action brings the instance state to *available* by updating the state attribute value. "Action Term" refers to the term of the Action's category identifier.

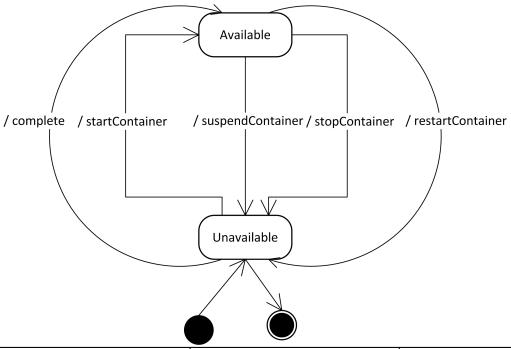
The *Database* resource attributes and actions are exposed by all Database type instances and are necessary to describe and handle the context of these instances (e.g. provisioning of a new instance, updating an existing instance, etc.). *Database* resource actions are used to manage these instances (e.g. start an instance, backup

an instance, etc.). Every defined action is identified by a *Category* instance using a */database/action#* categorization scheme.

The Container resource type

Table 5.3: Container type attributes.

Attribute	Type	Multiplicity	Mutability	Description
occi.container.name	String	0...1	Mutable	Name of the instance.
occi.container.version	String	0...1	Mutable	Version label of the instance.
occi.container. architecture	Enum {x86, x64}	0...1	Mutable	CPU architecture of the instance.
occi.container.state	Enum {available, unavailable}	1	Immutable	Current state of the instance.



Action Term	Target state	Attributes
StartContainer	Available	-
StopContainer	Unavailable	-
RestartContainer	Available	-
SuspendContainer	Available	-

Figure 5.3: State diagram and actions applicable to Container type instances

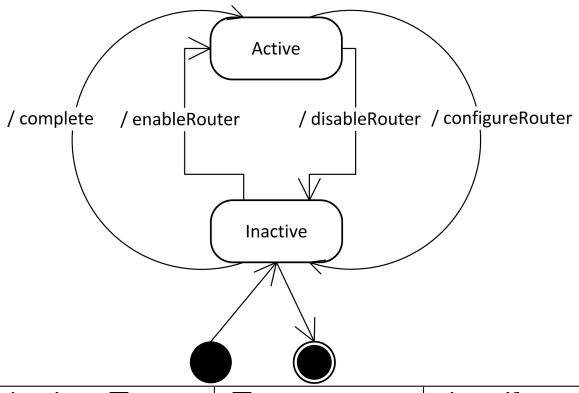
We defined the *Container* resource type that represents service containers, application servers and engines provisioned by PaaS to host and run applications (e.g. Apache Axis, Oracle GlassFish Server, etc.). We defined a set of attributes to describe a *Container* type (See Table 5.3) through its *Kind* instance (Container scheme) and a set of actions to manage the different *Container* instances into a PaaS (See Figure 5.3).

The defined attributes and actions are exposed by all *Container* type instances. The*state* attribute value indicates the current state value of an instance from the different predefined states that a *Container* instance can have (i.e. available, unavailable). The state value evolves after the execution of one of the actions according to the state diagram introduced in Figure 5.3 to keep a consistent and manageable set of instances.

The Router resource type

Table 5.4: Router type attributes.

Attribute	Type	Multiplicity	Mutability	Description
occi.router.name	String	0...1	Mutable	Name of the instance.
occi.router.version	String	0...1	Mutable	Version label of the instance.
occi.router. architecture	Enum {x86, x64}	0...1	Mutable	CPU architecture of the instance.
occi.router.state	Enum {active, inactive}	1	Immutable	Current state of the instance.



Action Term	Target state	Attributes
EnableRouter	Active	-
ConfigureRouter	None	Parameters
DisableRouter	Inactive	-

Figure 5.4: State diagram and actions applicable to Router type instances

The last platform resource type that we have defined is the *Router*. *Router* resources model message format transformations and routing systems provided by PaaS to route and deliver messages between PaaS components. *Router* entities are use-

ful where for example deployed applications on Cloud platforms are multi-tenant and/or service-based and requires then several (may be heterogeneous) containers to be hosted.

Table 5.4 describes the attributes that we have defined to describe the *Router* type through its *Kind* instance (Router scheme). The actions applicable to a *Router* instance and the diagram schematizing the evolution of the *state* value in relation with the execution of these actions are schematized in Figure 5.4. Each action is identified by a Category instance using a */router/action#* categorization scheme.

The DatabaseLink link type

To connect these defined platform resources, we modeled a set of platform link entities. These entities are extended from the OCCI core model *Link* base type. For example, to link a *Container* resource to a *Database* resource, we define the *DatabaseLink* type. This link enables a *Database* instance to be attached to a *Container* instance for applications interacting with a database system management for example (See Figure 5.5.).

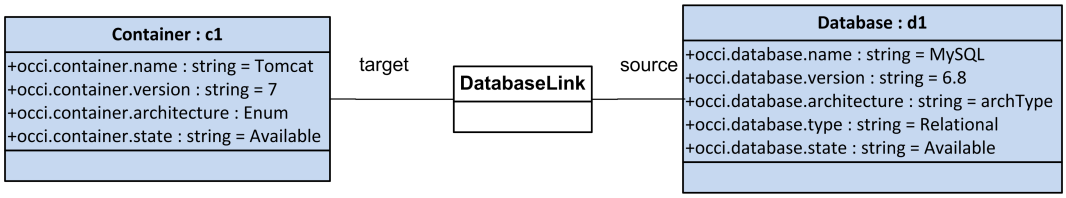


Figure 5.5: DatabaseLink type: A binding between Container and Database resources.

As examples of *DatabaseLink*, we can cite:

- .NET/Connector: which connects a .NET container to a MySQL instance,
- Mongo+Hadoop Connector: which connects a Hadoop server to a MongoDB instance.

A *DatabaseLink* instance can be set up between two or several *Container* and *Database* instances through the "Bind" action (see Table 5.5).

Table 5.5: Actions applicable to DatabaseLink instances.

Action	Attributes	Description
Bind	source, target	bind a Container instance source to a target Database instance.

The *ContainerLink* link type

We defined the *ContainerLink* type to enable connecting one or several homogeneous *Container* resources between them. By homogeneous, we mean same type container instances (see Figure 5.6.). For example, a multi-tenant J2EE application deployed onto two or more Apache Tomcat instances.

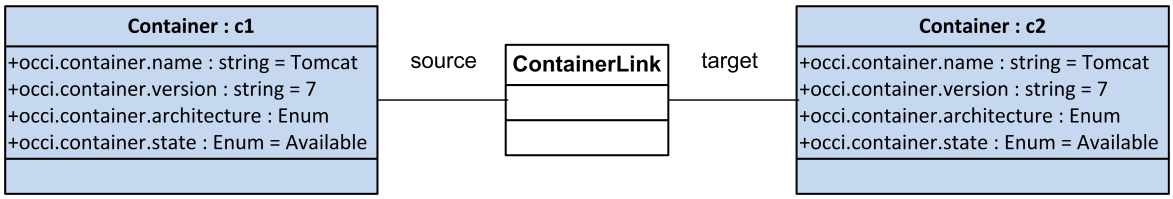


Figure 5.6: ContainerLink type: A connector between Container and Router resources.

The *RouterLink* link type

We defined the *RouterLink* type to enable connecting one or several heterogeneous *Container* resources to a *Router* resource. For example, a BPEL process deployed on an Apache ODE container instance and invoking as partner link a remote Web service deployed on an Apache Axis container instance during its execution (See Figure 5.7).

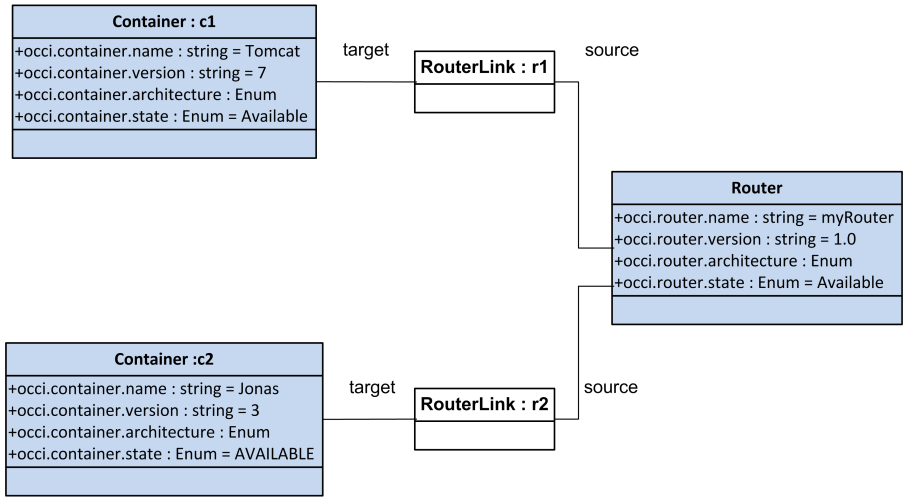


Figure 5.7: RouterLink type: A connector between several Container resources.

Examples of defined platform mixins

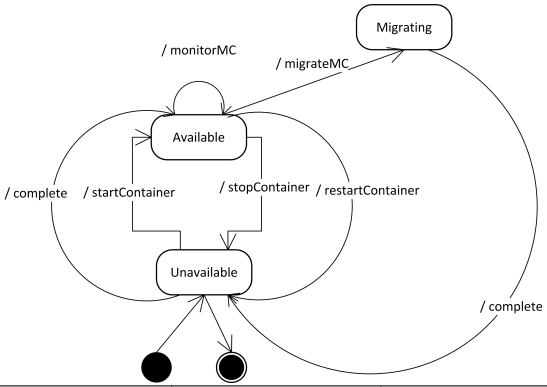
In addition to these defined platform resources and links, we can define platform *Mixin* if needed. Mixins are defined in order to support specific features and operations

offered by some PaaS and cannot be described by the main platform resources that we have defined. We consider for example the service micro-container that can be modeled as platform resource mixin. Indeed, in order to support the particular service micro-container capabilities (e.g. migration [73], monitoring [76], etc.), a correspondent mixin, which is a specialization of the *Container* resource type, is defined.

Table 5.6: Service micro-container mixin attributes.

Attribute	Type	Multiplicity	Mutability	Description
File_name	String	1	Mutable	MC file name (e.g. JAR name).
Requirements	Set of String	0...1	Mutable	Running requirements (e.g. JRE version, start command, etc.).
state	Enum {available, migrating, restarting, unavailable}	1	Immutable	Current state of the instance.

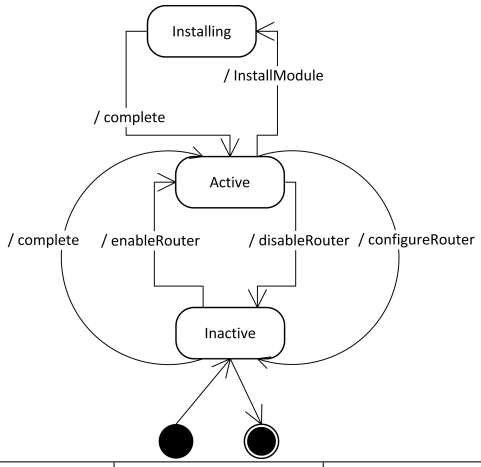
The service micro-container mixin attributes are listed in Table 5.6. Relative actions and associated state diagram are presented in Figure 5.8. Note that the micro-container mixin extends the state diagram of the *Container* resource type (See Figure 5.3) with an additional state i.e. *migrating* related to migration time (i.e. the *MigrateMC* action is being executed) from one host to another (See Section 4.4).



Action Term	Target state	Attributes
MonitorMC	Active	SLA document
MigrateMC	None	Target host
DuplicateMC	Active	-
ConsolidateMC	Inactive	-

Figure 5.8: State diagram and actions applicable to MicroContainer type instances

A second example of platform mixin might be a WSO2 ESB router mixin.



Action Term	Target state	Attributes
InstallModule	Available	module properties

Figure 5.9: State diagram and actions applicable to WSO2 router type instances

This mixin is specialized from the *Router* resource type. WSO2[3] is an Open Source Enterprise Service Bus (ESB) router which allows users to configure message routing, virtualization, intermediation, transformation, logging, task scheduling, load balancing, failover routing and event brokering.

WSO2 ESB design is specific and strongly extensible to allow integrating new modules if needed from a remote P2 repository (e.g. installing the Carbon UI Feature) [83]. To support these specific features, a WSO2 ESB router mixin was defined. The correspondent state diagram and relative action are defined in Figure 5.9.

Existing PaaS propose, apart platform resources, a set of technical functionalities and features (e.g. authentication, logging, metering, messaging, etc.) and even paid applications and services through marketplaces (e.g. Heroku add-ons [84], Cloud Foundry marketplace [85], etc.). These functionalities and services are also considered as platform resources. They are supported by our extension and can be modeled as mixins.

5.2.2 Application resources description model

In addition to the platform resource extension, we define an additionnal OCCI-based application extension. The purpose of this model is to describe an application (i.e. any computer software or program) that can be hosted and executed by a PaaS using defined platform resources. Quite like our defined platform resource types, the application resource types extends OCCI core types. Both applications resources and the links between them are respectively derived from *Resource* and *Link* types of the

[3] wso2.com

OCCI core model (See Figure 5.10). Table 5.7 describes the *Kind* instances defined for each one of the application *Resource* or *Link* sub-types.

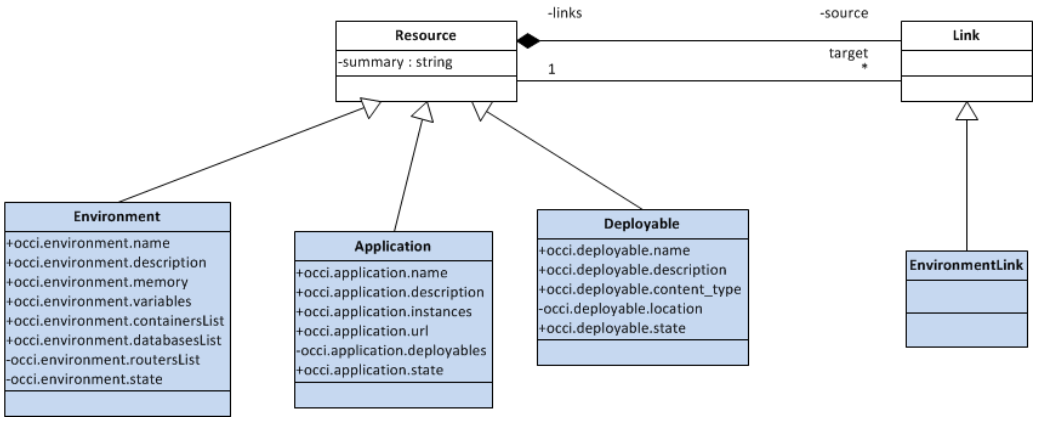


Figure 5.10: Overview of the defined OCCI application types.

The defined OCCI application types are:

- *Environment* which represents a set of "settings" needed to host and run an *Application* (e.g. runtime, framework, message queue, etc.),
- *Application* which is the software or program that can be deployed on top of a PaaS (WAR file, Ruby program, etc.).
- *Deployable* which represents the *Application* deployables (e.g. sources archives, etc.),
- *EnvironmentLink* which connects an *Application* to an *Environment*.

Table 5.7: The kind instances defined for the application subtypes of Resources and related Links.

Term	Scheme	Title	Related Kind
Environment	$< schema > /application\#$	Environment Resource	$< schema > /core\#resource$
Application	$< schema > /application\#$	Application Resource	$< schema > /core\#resource$
Deployable	$< schema > /application\#$	Deployable Resource	$< schema > /core\#resource$
EnvironmentLink	$< schema > /application\#$	Environment Link	$< schema > /core\#link$

The *Environment* resource type

The *Environment* resource models a set of configurations and settings of the platform resources (e.g. *Container* resources, *Database* resources, *DatabaseLink* resources, etc.)

and needed to host and run applications on PaaS. *Environment* resource includes, among others, the needed runtime (e.g. java 7, java 6, ruby, etc.), the needed frameworks/containers (e.g. spring, tomcat, ruby, etc.) and optionally needed provider services (e.g. monitoring, messaging, etc.). The OCCI attributes defined for an *Environment* resource through its *Kind* resource instance are listed in Table 5.8. We defined also an action exposed by all *Environment* type instances in order to update an already existing environment (See Table 5.9).

Table 5.8: Environment resource type attributes.

Attribute	Type	Multiplicity	Mutability	Description
occi.environment.name	String	0..1	Mutable	Name of the instance.
occi.environment.description	String	0..1	Mutable	Human readable description of the instance.
occi.environment.memory	Float, 10^9 (GiB)	0..1	Mutable	RAM allocated to the instance.
occi.environment.variables	Set of (var, value)	0..1	Mutable	Environment variables associated to the instance.
occi.environment.containersList	Set of URIs	0..1	Mutable	Set of URIs of Container instances associated to the instance.
occi.environment.databasesList	Set of URIs	0..1	Mutable	Set of URIs of Database instances associated to the instance.
occi.environment.routersList	Set of URIs	0..1	Mutable	Set of URIs of Router instances associated to the instance.
occi.environment.databasesLink	Set of URIs	0..1	Mutable	Set of URIs of DatabaseLink instances associated to the instance.
occi.environment.state	Enum {available, unavailable}	1	Immutable	Current state of the instance.

The *Application* resource type

The *Application* resource type models any computer software or program that can be deployed on top of a PaaS. Defined *Environment* resource type instance enables

Table 5.9: Actions applicable to Environment type instances.

Action Term	Target state	Attributes
Update	None	Platform resources list

hosting and executing one or more *Application* resource type instance(s). This hosting *Environment* is set up thanks to the instantiation and the configuration of necessary platform resources, links and mixins.

To deploy an *Application*, the end-user specifies a set of properties (e.g. application name, application description, etc.). Moreover, if the target PaaS supports the management of multiple instances (e.g. Cloud Foundry), the user can specify the desired number of active instances to ensure application scalability and availability (See Table 5.10).

Once deployed, the *Application* can be invoked and executed through its public URL provided by the target PaaS. To manage the different *Application* instances, we defined a set of actions (See Table 5.11). These actions allow to start (respectively stop) an already deployed application in a hosting *Environment* and then to update its state value to *available* (respectively *unavailable*). The update action allow reload an application after setting and/or *Deployables* changes.

Table 5.10: Application type attributes.

Attribute	Type	Multiplicity	Mutability	Description
occi.application. name	String	0..1	Mutable	Name of the instance.
occi.application. description	String	0..1	Mutable	Human readable description of the instance.
occi.application. instances	integer	1..N	Mutable	Number of the instance copies.
occi.application. url	URL	0..1	Mutable	The public URL associated to the instance.
occi.application. deployables	Set of URIs	0..1	Mutable	Set of URIs of Deployable associated to the instance.
occi.application. state	Enum {started, stopped}	1	Immutable	Current state of the instance.

Table 5.11: Actions applicable to Application type instances.

Action Term	Target state	Attributes
Update	None	Application description
Start	started	-
Stop	stopped	-
Restart	started	-

The *Deployable* resource type

The *Deployable* type models the *Application* source archives. By deployables, we mean all necessary artifacts (e.g. ZIP file, EAR file, etc.), configuration files (e.g. Chef script, etc) and/or deployment descriptors (e.g. XML file, DAT file, etc.) needed to carry out the application deployment (see Table 5.12).

The end-user can upgrade *Deployable* instances to apply new source updates for example through the "update" action (see Table 5.13).

Table 5.12: Deployable type attributes.

Attribute	Type	Multiplicity	Mutability	Description
occi.deployable.name	String	0..1	Mutable	Name of the instance.
occi.deployable.description	String	0..1	Mutable	Human readable description of the instance.
occi.deployable.content_type	Enum {artifact, war, jar, ear}	0..1	Mutable	Archive types of the instance.
occi.deployable.location	URL	0..1	Mutable	Location of the artifact associated to the instance. It can be a file path or a logical Name.
occi.deployable.state	Enum {available, unavailable}	1	Immutable	Current state of the instance.

Table 5.13: Actions applicable to Deployable type instances.

Action Term	Target state	Attributes
Update	None	Artifacts

The *EnvironmentLink* link type

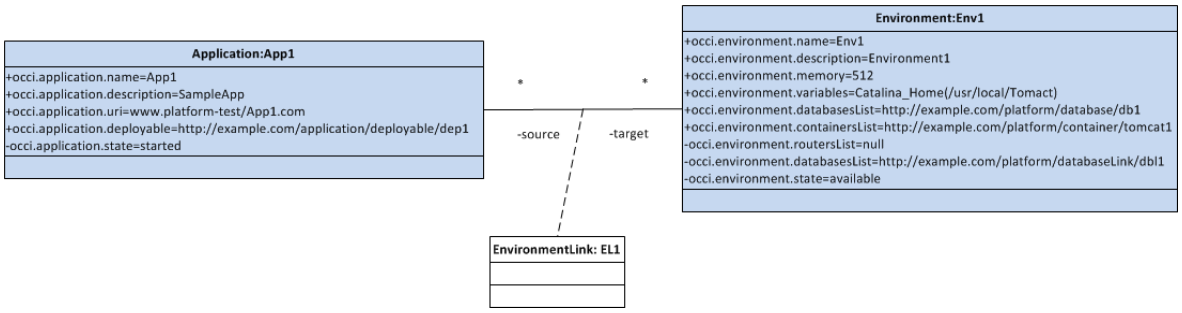


Figure 5.11: EnvironmentLink type: A connector between Application and Environment resources.

In order to connect an *Application* resource to a hosting *Environment* resource, we defined the *EnvironmentLink* type. *EnvironmentLink* resources are extended from the OCCI core model *Link* base type. Concretely, to deploy an *Application* on top of an *Environment*, one must instantiate an *EnvironmentLink* which links them (See Figure 5.11).

5.3 COAPS API specifications

In this Section, we detail COAPS API specifications. COAPS is a REST-based and OCCI-compliant API implementing our introduced description model (See Section 6.2.3 for COAPS implementation details). The choice of the REST architecture was motivated by the type of existing Cloud platforms APIs interacting with COAPS wich are almost all REST-based. COAPS exposes a set of generic interfaces for applications and PaaS resources provisioning and management and is based on a proxy system that we have designed to provide appropriate implementations of these interfaces when interacting with existing Cloud platforms. The full version of the COAPS API specifications is available at [86].

5.3.1 COAPS generic interfaces overview

The COAPS generic interfaces are classified into two resource management packages:

1. The Environment management package which provides COAPS generic operations to create and manage *Environment* resources,

2. The Application management package which provides generic COAPS operations to create and manage *Application* resources.

A resource-based representation of the proposed environment management operations is provided in Figure 5.12. Each box represents an *Environment* resource (or

sub-resource), the title text (e.g. `/environment`, `/environment/envId`, etc.) represents the resource identifier and the body text lists the offered operations by this resource (e.g. `FindEnvironments`, `CreateEnvironment`, etc.) and its associated REST methods (e.g. GET, POST, etc.).

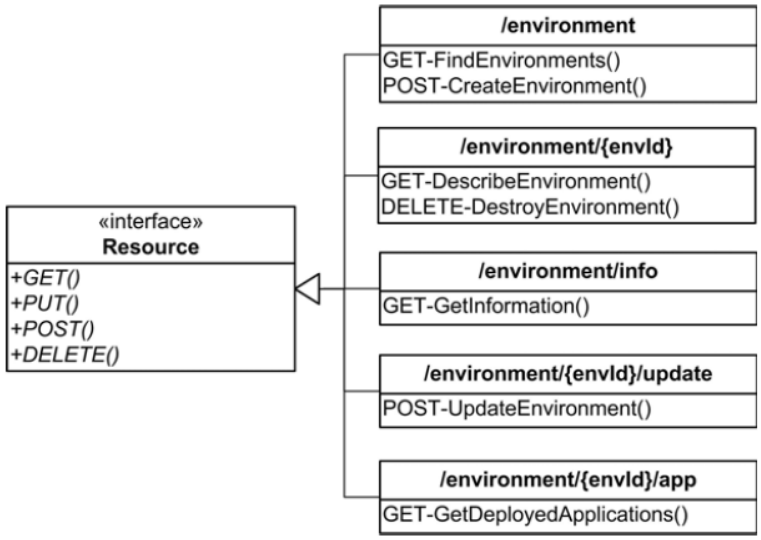


Figure 5.12: The COAPS API environment management operations.

In COAPS specifications, we consider the basic operations for an application's environment creation and management. The environment management resource offers the following operations:

- *Create Environment*: creates a new environment using the *paas_ environment* element of the manifest. The operation returns, among others, an environment ID.
- *Update Environment*: updates an existing environment. An environment ID must be provided and the updates must be specified in a new manifest.
- *Destroy/Describe Environment*: destroys/describes an environment given its ID.
- *Find Environments*: lists all available environments.
- *Get Deployed Applications*: lists all deployed applications in an environment given its ID.
- *Get information*: lists the runtimes, frameworks and services supported by the targeted PaaS.

As for the environment management resource, we consider the basic operations for an application provisioning and management. Our application resource management package is represented in Figure 5.13.

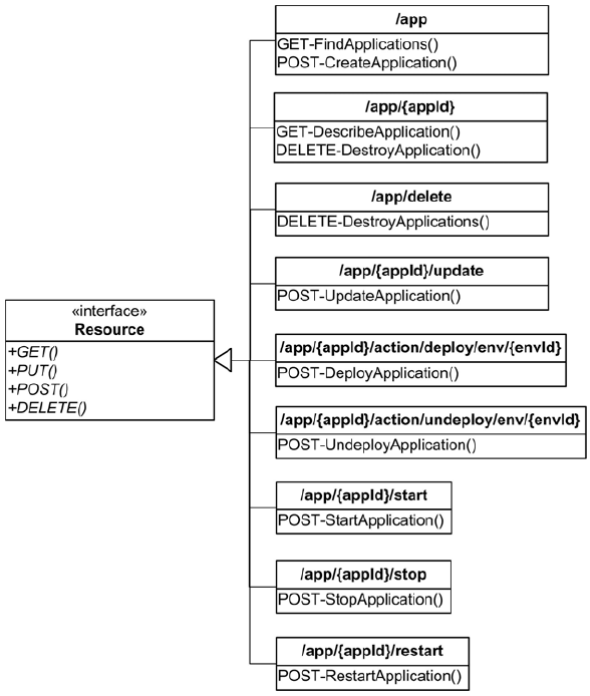


Figure 5.13: The COAPS API application management operations.

The application resource management package offers the following operations:

- *Create Application*: creates a new application using the application description in the manifest. The operation returns, among others, an application ID.
- *Deploy Application*: deploys an application identified by its ID on an existing environment identified by its environment ID.
- *Start/Stop/Restart/Un-deploy/Destroy Application*: starts/stops/restarts/un-deploys/destroys a deployed application given its ID.
- *Update Application*: updates an existing application. The application ID must be provided and the updates have to be specified in a new manifest.
- *Describe Application*: returns an application description given its ID.
- *Find Applications*: lists the available applications.
- *Destroy Applications*: destroys all existing applications.

Some of COAPS operations (e.g. `CreateEnvironment`, `UpdateEnvironment`, `CreateApplication`, etc.) require attached manifests to well describe the resources to manage. COAPS manifests are XML-based documents that describe a given *Application* or its hosting *Environment* resources according to our introduced description model.

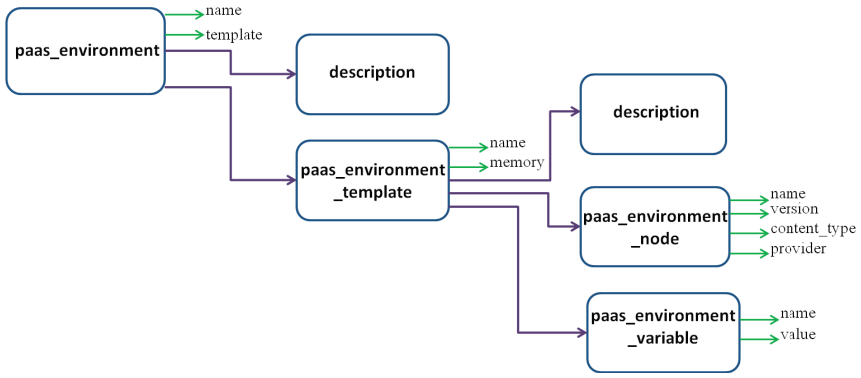


Figure 5.14: XML schema of an Environment resource manifest.

Figure 5.14 details the XML schema of an *Environment* resource. An environment manifest contains a *description* of an *Environment* resource and details about the *paas-environment-template* from which it is instantiated. A *paas-environment-template* element is characterized by a name, a memory size, a description, a list of *paas-environment-node* elements and optionally a list of *paas-environment-variable* elements if needed. A *paas-environment-node* element provides information about a platform resource to provision according to our description model (*content_type* attribute). A *paas-environment-variable* element allow the specification of a set of variables necessary for the configuration and execution of requested *paas-environment-node* elements (e.g. an environment variable to configure a container node).

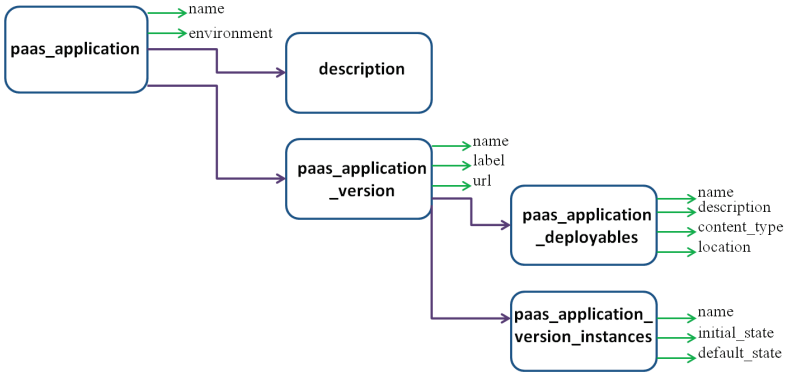


Figure 5.15: XML schema of an Application resource manifest.

Figure 5.15 details the XML schema of an *Application* resource. An *Application* manifest contains a *description* of an *Application* resource and details about the *paas-application-version* of the application to deploy/manage. A *paas-application-version* is characterized by a name, a label, a public uri to provide to the PaaS solution to execute the application once deployed, a *paas-application-deployable* element and description of one or more required *paas-application-version-instance*. The *paas-application-deployable* element describes the application source archives format (e.g. WAR, EAR, etc.) and their location for the upload processing.

5.3.2 COAPS proxy system

To generalize and facilitate the use of our solution, we designed a PaaS proxy system that allows users to adapt and configure COAPS generic interfaces in order to provision resources from any existing PaaS.

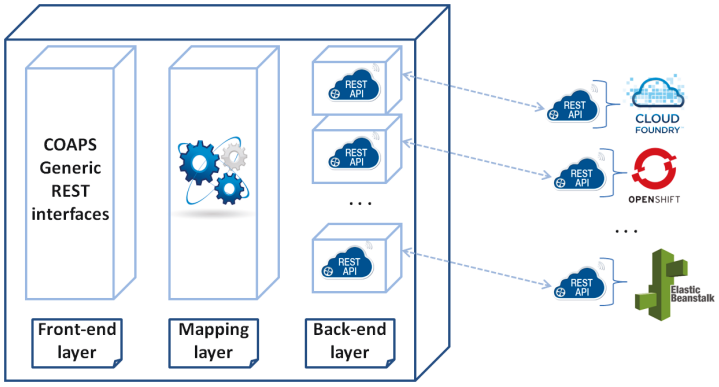


Figure 5.16: COAPS proxy system.

The COAPS proxy architecture is detailed in Figure 5.16. It consists of three parts:

- Front-end layer which contains the COAPS generic interfaces. These interfaces expose the generic RESTful operations to manage and provision applications and hosting environments,
- Back-end layer which represents the PaaS providers interface. This interface exposes the proprietary API operations of a specific PaaS,
- Mapping layer which represents the middle layer of the proxy. This layer ensures the mapping between the COAPS generic operations and the proprietary PaaS API operations.

To define a new COAPS implementation to provision resources from a given PaaS, one can simply instantiate its proxy after implementing its correspondent middle layer

to map between the COAPS generic interfaces and the proprietary actions exposed by the selected PaaS API. Several COAPS interfaces can be coupled to a single PaaS operation if needed (e.g. POST Deployables and POST Application operations for Cloud Foundry implementation) and/or some COAPS operations can be ignored (and then not implemented) for the case where there are not supported by a specific PaaS (e.g. POST RouterLink operation for Cloud Foundry implementation insofar as Cloud Foundry manages itself the routage between DEA components). Cloud Foundry DEAs are the components which contains and manages the embedded service containers. More details about current available COAPS implementations are provided in Section 6.2.3.2.

5.3.3 Deployment of service-based applications using COAPS

Generally, service-based applications deployment through COAPS API is performed according to the scenario steps detailed in Figure 5.17.

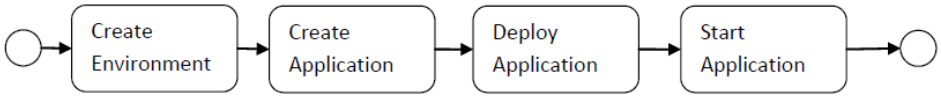


Figure 5.17: Provisioning applications scenario steps through COAPS API.

These steps represents generic operations of COAPS detailed in Section 5.3.1. Thanks to COAPS, to provision an application in a PaaS, we follow the same provisioning scenario, the same API operations and the same resources descriptors whatever is the target Cloud platform.

5.4 Examples of service-based applications deployment

As illustrative examples, we propose to deploy the shop process and ComputePrice micro-containers obtained after slicing the applications and packaging their services. The deployment is performed in both IaaS and PaaS.

For an IaaS deployment, we used the Network and Cloud Federation (NCF) experimental platform deployed at Télécom SudParis. We used OpenNebula IaaS manager to instantiate hosting VMs. For a PaaS deployment, we used Cloud Foundry PaaS.

In the following, we describe the work that we have done to deploy and execute shop process and ComputePrice application service micro-containers.

5.4.1 Deployment of shop process

All needed services to deploy the shop process are packaged in micro-containers. There is no more additional modules (e.g. remote Web services, external partner links)

interacting with the BPEL process to provision in the Cloud. The deployment is performed in both NCF Cloud infrastructure and Cloud Foundry PaaS.

5.4.1.1 Deployment of shop process in NCF infrastructure

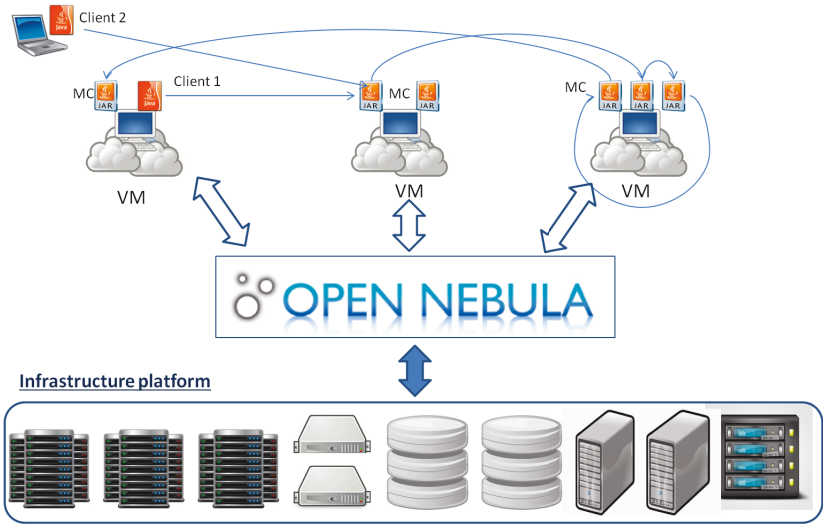


Figure 5.18: Deployment of the shop process in NCF infrastructure.

Service micro-containers deployment in NCF Cloud infrastructure is performed as follows:

1. Instantiate the hosting VMs using OpenNebula IaaS manager,
2. Install the Java Virtual Machine (JRE) in the hosting VMs,
3. Upload the JAR files on the VMs using SSH protocol,
4. Run the JAR files to start listening to client requests.

An overview of the deployment process is illustrated in Figure 5.18. The placement choice of the service micro-containers is arbitrary and not dealt in our work.

5.4.1.2 Deployment of shop process in Cloud Foundry

An overview of the deployment process is illustrated in Figure 5.19. To deploy the shop process in Cloud Foundry, we use the correspondent COAPS implementation that maps our generic interfaces with the Cloud Foundry proprietary operations and we follow the scenario introduced in Section 5.3.3.

To create an *Environment* resource, we call the `CreateEnvironment` operation and we provide the description of the required *Environment* to provision in a manifest

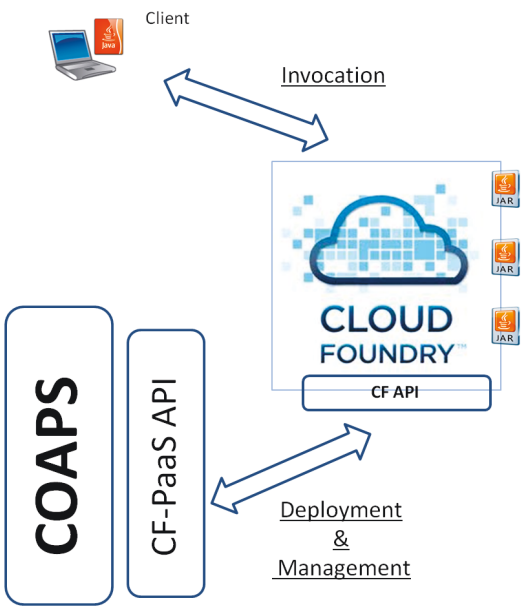


Figure 5.19: Deployment of the shop process in Cloud Foundry.

attached to the POST request (See Listing 5.1). This operation returns back an *envId*. Since we deploy exclusively micro-containers, we specify only the description of the JVM to allocate as *paas_environment_node* (line 6).

Listing 5.1: The Environment resource manifest.

```
1 <?xml version="1.0" encoding="UTF8"?>
2 <paas_environment name="MC_Env" template="MC_EnvTemp">
3     <description>This manifest describes the hosting environment of an MC.
          </description>
4     <paas_environment_template name="MC_EnvTemp" memory="512">
5         <description>MCEnvironmentTemplate</description>
6         <paas_environment_node content_type="container" name="jvm" version="
              7" provider="CF"/>
7     </paas_environment_template>
8 </paas_environment>
```

To create an *Application* resource, we call the `CreateApplication` operation and we provide the description of the *Application* to deploy, the content_type and the location of its *Deployables* in a manifest attached to the POST request. This operation returns back an *appId*. In this operation, our micro-containers are assimilated to standalone and autonomous applications and are deployed one at a time. Listing 5.2 provides description of micro-container that hosts the first service of the process (*service*1).

Listing 5.2: The Application resource manifest.

```
1 <?xml version="1.0" encoding="UTF8"?>
```

```
2 <paas_application name="Service1" environment="MC_Env">
3     <description>Java Service packaged in MC.</description>
4     <paas_application_version name="version1.0" label="1.0" url="service1.
          cfapps.io">
5         <paas_application_deployable name="MC1.jar" description="Jar_file"
              content_type="artifact" location="/home/yangui/MCs"/>
6         <paas_application_version_instance name="Instance1" initial_state="1
              " default_instance="true"/>
7         <paas_application_version_instance name="Instance2" initial_state="1
              " default_instance="false"/>
8     </paas_application_version>
9 </paas_application>
```

After that, we call the `DeployApplication` operation to process the uploading of the sources and deployment of the application in the target PaaS provider. Specifically, this operation requires the already created *envId* and *appId* to perform the link between them (*EnvironmentLink resource*).

Once the application is deployed, we call the `StartApplication` operation to run it and make it available to prospective request calls

5.4.2 Deployment of ComputePrice application

The ComputePrice SCA-based application has an external component in the form of remote Web service (*DetermineTaxRate* component) and a database instance (See Section 2.1.4). So, provisioning the ComputePrice application requires the deployment of: (i) its correspondent micro-containers, (ii) *DetermineTaxRate* Web service on a reachable Tomcat instance and (iii) the used database instance. The provisioning is performed in both NCF infrastructure and Cloud Foundry PaaS.

5.4.2.1 Deployment of ComputePrice application in NCF infrastructure

An overview of the performed deployment is detailed in Figure 5.20. The ComputePrice application provisioning in NCF infrastructure is performed as follows:

1. Instantiate the hosting VMs using OpenNebula IaaS manager,
2. Install the JRE in the hosting VMs,
3. Install a Tomcat container instance in a VM and deploy the *DetermineTaxRate* on it,
4. Install MySQL DBMS and copy the database records on it,
5. Setup the connector[4] between MySQL and Tomcat intances,
6. Upload the JAR files in the VMs using SSH protocol,
7. Run the JAR files to start listening to client requests.

[4] http://dev.mysql.com/downloads/connector/j/

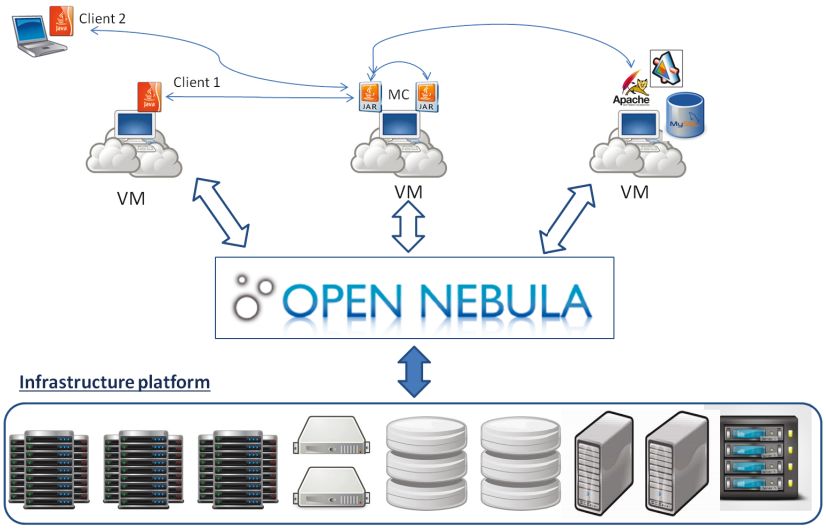


Figure 5.20: Deploying ComputePrice application in NCF infrastructure.

5.4.2.2 Deployment of ComputePrice application in Cloud Foundry

The deployment of ComputePrice application in Cloud Foundry is carried out according to the scenario introduced in Section 5.3.3. For this deployment, in addition to the service micro-containers deployment, we need to provision a tomcat server and MySQL instances from Cloud Foundry for deploying the *DetermineTaxRate* Web service and the database used by the application.

Listing 5.3: The Environment resource manifest.

```
1 <?xml version="1.0" encoding="UTF8"?>
2 <paas_environment name="JavaWeb_Env" template="JavaWeb_EnvTemp">
3     <description>This manifest describes the hosting environment for a WS.
          </description>
4     <paas_environment_template name="JavaWeb_EnvTemp" memory="1024">
5         <description>JavaWebEnvironmentTemplate</description>
6         <paas_environment_node content_type="container" name="Tomcat"
              version="6" provider="CF"/>
7         <paas_environment_node content_type="database" name="MySQL" version=
              "7" provider="CF"/>
8         <paas_environment_node content_type="databaseLink" name="
              database_Link"  provider="CF"/>
9     </paas_environment_template>
10 </paas_environment>
```

Specifically, for the micro-containers deployment through COAPS Cloud Foundry implementation, we create firstly the hosting environment (*MC_Env*) instantiated from *MC_EnvTemp* template. After that, we create a set of applications (one per micro-container) before deploying them in *MC_Env*.

To provision the tomcat server and the database instance, we create a novel appropriate environment called *JavaWeb_Env* (See Listing 5.3). Then, we created a novel application associated to the *DetermineTaxRate* Web service and we deploy it *JavaWeb_Env* environment (See Listing 5.4).

Listing 5.4: The Application resource manifest.

```
1 <?xml version="1.0" encoding="UTF8"?>
2 <paas_application name="DetermineTaxRate" environment="JavaWeb_Env">
3     <description>DetermineTaxRate WS.</description>
4     <paas_application_version name="version1.0" label="1.0" url="
          DetermineTaxRate.cfapps.io">
5         <paas_application_deployable name="DetermineTaxRate.WAR" description
              ="WAR_file" content_type="artifact" location="/home/yangui/WS"/>
6         <paas_application_version_instance name="Instance1" initial_state="1
              " default_instance="true"/>
7     </paas_application_version>
8 </paas_application>
```

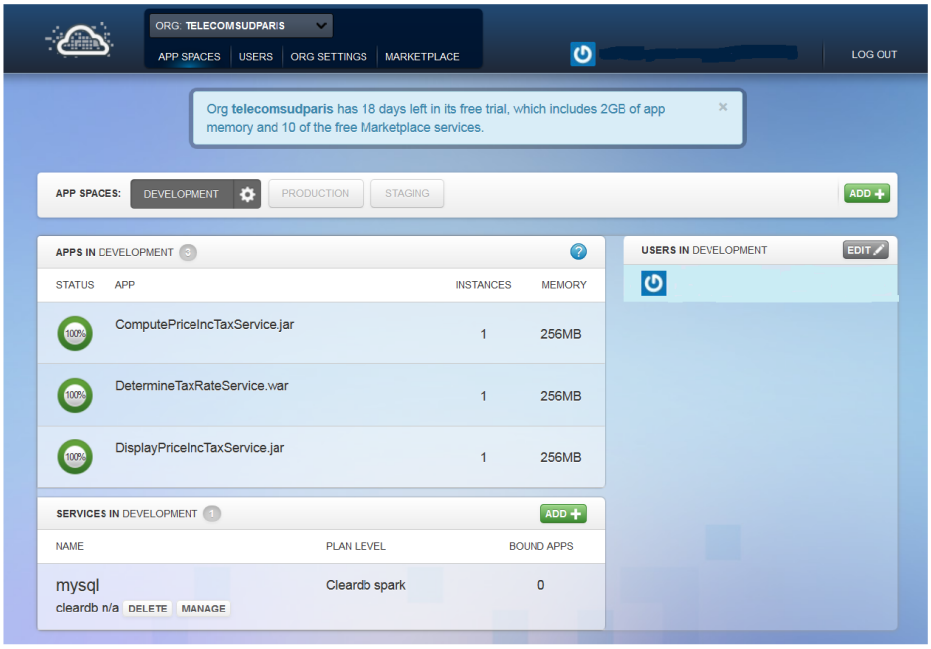


Figure 5.21: Cloud Foundry Web graphic console screenshot showing deployed ComputePrice application services.

Figure 5.21 presents a screenshot of the Cloud Foundry Web graphic console[5] showing our three services deployed in Cloud Foundry. Generally, when deploying in PaaS through COAPS, we have only to provide required platform resources description. In fact, instantiation and parameterization of these resource (e.g. Tomcat instance in-

[5] console.run.pivotal.io/

stallation, MySQL installation, jdbc-connector setup, etc.) are delegated to the PaaS contrary to what we have done to provision the same application in an IaaS. Finally, it should be noted that to deploy the ComputePrice application in another PaaS (e.g. OpenShift), we use exactly the same manifests and the same operations.

5.5 Conclusion

In this Chapter, we presented and detailed the third step of the SPD approach i.e. deploying the packaged services in micro-containers in the Cloud. Both deployment in Cloud infrastructures and Cloud platforms are treated. Briefly, the deployment in IaaS consists on instantiating VMs and uploads the service micro-containers on them before starting their execution as standalone applications. For the deployment in a PaaS, we defined a generic description model for applications and PaaS resources. Our proposed description model is based on the OCCI standard. According to OCCI, each identified PaaS resource is characterized by a set of attributes, management actions and associate lifecycle. Based on this model, we are able to describe, provision and manage a given PaaS resource in an unified way whatever is the target Cloud platform. We also performed a PaaS-independent REST API called COAPS implementing this model to process applications deployment and management on target PaaS. We presented and commented the COAPS specifications.

In the last Section of the Chapter, we illustrated our findings by showing the deployment procedures for the shop process and ComputePrice applications in both NCF infrastructure and Cloud Foundry PaaS.

In the next Chapter, we detail and discuss the implementation details of each one of the SPD approach steps.

Chapter 6

Implementation & Experiments

Contents

6.1 Introduction

In this chapter, we present the implementations we have done to realize the SPD approach, and the experiments we have made to evaluate the effectiveness of our developed tools. Our goal is to prove that our approach is feasible and effective in real uses cases. To that end, we have implemented each step of our defined SPD approach.

To implement the first step of the SPD approach i.e. slicing a service-based application in a set of elementary services, we developed tools implementing our algorithms introduced in Chapter 3. The developed tools support slicing and aggregating BPEL-based processes and SCA-based applications.

To implement the second step of the SPD approach i.e. packaging sliced services in appropriate micro-containers, we implement the packaging framework that allows the service micro-containers building from the framework generic modules (See Chapter 4).

To implement the third step of the SPD approach i.e. deploying the micro-containers in a target Cloud environment, we develop a REST API implementing our proposed applications and PaaS resources description model (See Chapter 5). Our performed API is a PaaS-independent solution which enables provisioning and managing services and applications in existing Cloud platforms through appropriate implementation of its generic operations.

This Chapter is organized as follows: we describe development details and used technologies to implement each step of the SPD approach in Section 6.2. Then, we present the experimentations that we have performed to (1) highlight classical service container limitations in Cloud environments and (2) evaluate our service micro-container performances versus classical service containers in Section 6.3. Finally, we present a realistic use case resuming our findings in Section 6.4.

6.2 Implementation

The implementation details of our developed tools are detailed in the following Section. For each step of the SPD approach, we developed tools that implements algorithms and architectures that we have introduced. Applications slicing and services aggregation tools are detailed in Section 6.2.1. The packaging framework implementation is detailed in Section 6.2.2. COAPS API implementation is detailed in Section 6.2.3.

6.2.1 Application slicers and services aggregation tools

We develop service-based application slicers and services aggregators implementing our algorithms introduced in Chapter 3. Our tools support BPEL-based processes and SCA-based applications processing. For BPEL-based processes, we develop the *BPEL2Java* tool that slices a given BPEL according to Algorithm 1 before generating and aggregating the java code of each obtained subnet. For SCA-based applications, we develop the *SCA2Java* tool that slices a given SCA application according to Algorithm 3 before aggregating the java code of obtained services.

6.2.1.1 BPEL2Java tool

For BPEL-based processes slicing, we follow the following methodology:

1. Generate a Petri net graph from the BPEL to deploy,
2. Slice the Petri net into a set of dependent WF-nets,
3. Sort the subnets to obtain the whole execution chain equivalent to the initial BPEL,

4. Generate and aggregate services code corresponding to the WF-nets.

Table 6.1: Examples of defined transformation rules of basic BPEL activities to Java instructions.

BPEL activity	**Equivalent Java instruction(s)**
Invoke:one way	Java asynchronous remote call
Invoke	Java synchronous remote call
Assign	Java assignment instruction
Reply	Java *return()* instruction
Receive	Java method endpoint
Fault	Java *catch* sequence
Compensate	Java *finally* sequence
Wait	Java *wait()* instruction
Terminate	Java *return(0)* instruction
Exit	Java *return(-1)* instruction

The BPEL to Petri net transformation is performed by the *BPEL2PN* tool (See Section 3.2.4). Then, we execute Algorithm 1 to slice the Petri net. This algorithm provides a set of dependent WF-net services. The execution order of these WF-nets is provided by the execution of Algorithm 2 based on the dependency function *Dep* computed by Algorithm 1. Assuming that each subnet corresponds to a fragment of the *.bpel* process descriptor, we developed a java tool called *BPEL2Java* ensuring java service code generation for each sliced subnet based on the following fundamentals:

- Each Petri Net transition is equivalent to its correspondent BPEL activity,
- None of the Petri net places has equivalent in BPEL. Places make sense only in a Petri net network.

BPEL2Java sources are available at [87]. These sources are composed of 11 packages, 155 classes and 28131 instructions. The java code generation is based on transformation rules that we have defined to be able to convert a BPEL activity to an equivalent java sequence instructions (e.g. an *Invoke:oneway* BPEL activity will be transformed to a Java asynchronous remote call, an *Invoke* activity to a Java

sychronous remote call, an *Assign* activity to a Java assignment instruction and so on). Table 6.1 details a non-exhaustive list of transformation rules that we have defined.

To support the *Invoke* activity, we add to services code a parameterized generic client stemming from a generic soap client code[1]. The parameters provided to each client are based on information on the *.bpel* process descriptor and its prospective related *.wsdl* partner link descriptors. Furthermore, the structured BPEL activities (e.g. *IF*, *Else*, *While*, *Switch*, *Case*, etc.) are still the same in Java. The java source code generation is performed by the use of the *Codemodel*[2] tool. *Codemodel* is a Java library providing a way to generate Java programs using appropriate packages.

It should be noted that some information are likely to be lost when we generate the Petri net from the *.bpel* descriptor. Indeed, the choice of execution branch is a non-deterministic choice in a Petri net. Thus, structured BPEL activities (e.g. *IF*, *While*, etc.) can not be represented using a Petri net. To find this kind of information, we annotate the Petri Net elements with these tests and/or conditions.

6.2.1.2 SCA2java tool

Table 6.2: Examples of defined transformation rules of SCA annotations to Java instructions.

SCA annotation	Equivalent Java instruction(s)
@Property	Java type
@Reference	local java call
@Remotable	remote java call

We developed a tool called *SCA2java* implementing Algorithm 3 [88]. The tool sources are available at [87]. These sources are composed of 22 packages, 79 classes and 27339 instructions. The execution of *SCA2java* is schematized in Figure 6.1. The tool parses the *.composite* of a given SCA-based application descriptor using Eclipse modeling framework (EMF[3]) and initializes the *serviceList* structure. Each element of *serviceList* is loaded by an application component and its sub-elements (e.g. service, reference, properties, binding, etc.). After that, for each element of *serviceList*, we create a correspondent java service. Each generated service represents an implementation of one of the SCA-based application components. These services are composed of two classes: an *interface* class and an implementation class of this interface. The source code of these services is copied from the initial SCA project. Some aggregation tasks are also performed in the services source code to ensure their

[1] https://github.com/impactcentre/iif-generic-soap-client
[2] codemodel.java.net/
[3] http://www.eclipse.org/modeling/emf/

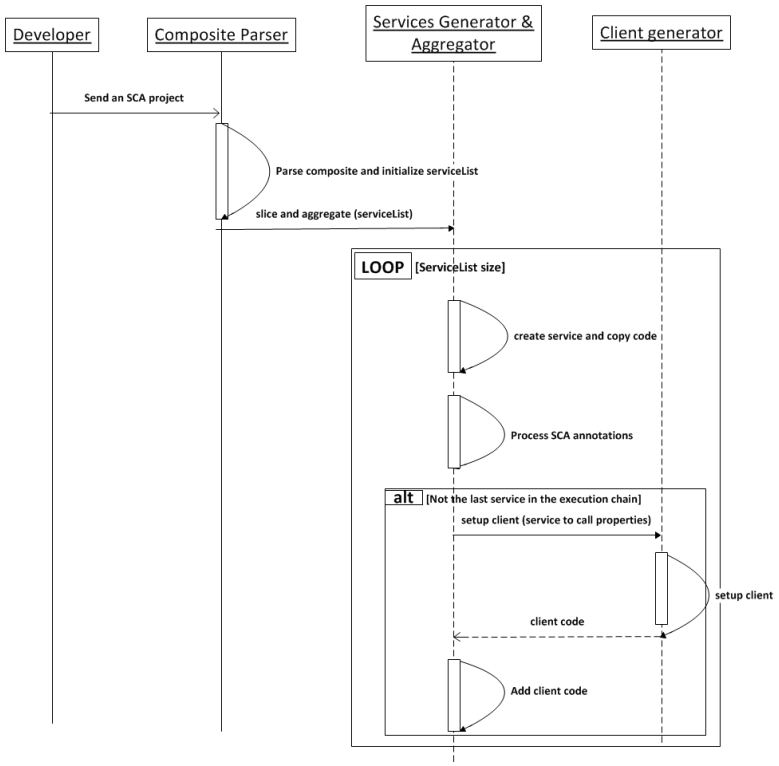


Figure 6.1: *SCA2java* execution process sequence diagram.

validity and compliance with java service specifications. Among the major changes that we operate, we can cite the transformations of the SCA annotations to a standard java code. Table 6.2 details a non-exhaustive list of transformation rules from SCA annotations to standard java code that we have defined.

The remote java calls instead of *@Remotable* annotation are performed by injecting the code of parameterized generic soap client[4]. The client properties (i.e. name of the target service, name of the method to invoke, input parameters number and types) are determined from the *.composite* SCA-application descriptor and prospective *.wsdl* remote services descriptors.

6.2.2 Packaging framework tool

We implemented the packaging framework using java according to the system architecture introduced in Section 4.2. The generic framework modules are implemented progressively as java packages and are integrated then in the packaging process when they are linked to the processing module. By doing so, we guarantee that these modules are plugables. In fact, adding new communication protocols or programming

[4]https://github.com/impactcentre/iif-generic-soap-client

languages support consists in adding the correspondent components in the packaging framework generic packages.

Before starting the packaging process, the developer can specify some settings (e.g. location of the generated micro-container, name of the generated micro-container, listening port value, etc.) through a *.properties* file representing the deployment descriptor of the micro-container. In addition to that, the packaging framework provides the possibility to package in micro-containers services with specific requirements (e.g. specific librairies, configuration files, etc.). To perform this, the processing module adapts the service invocation package using the JavaAssist[5] tool.

Figure 6.2 schematizes the performed packaging framework tool modules and the service micro-container generation process. The developed sources are composed of 10 packages, 155 classes and 4543 instructions.

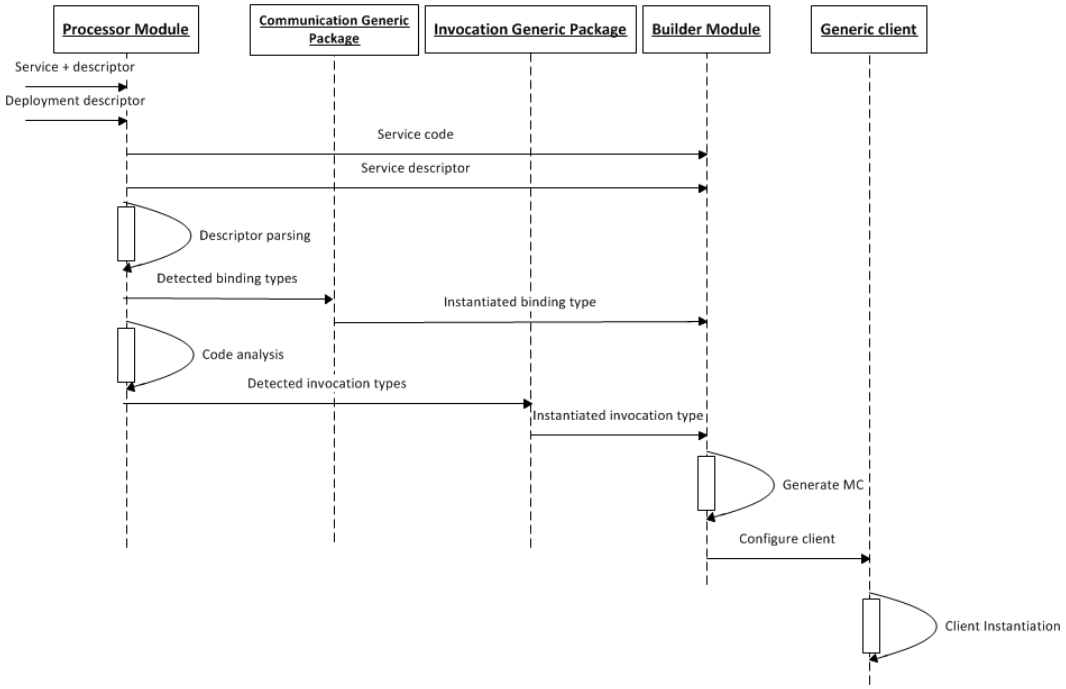


Figure 6.2: Service packaging and micro-container generation sequence diagram.

The *Processor module* analyzes the service code, parses its associated descriptor to determine the service binding types and sends it to the *Communication Generic Package* to instantiate the appropriate module implementing these bindings. The same principle is used by the *Invocation Generic Package* to instantiate the appropriate service invocation module. The instantiated modules are sent to the *Builder module* which is responsible of generating the micro-container packaging the service,

[5] javassist.org/

the instantiated modules and a set of generic service container classes based on information provided in the deployment descriptor. Generated micro-containers are autonomous and standalone applications packaged as running Java ARchives (JAR file) thanks to One-Jar tool[6]. The information about the main class to execute in the micro-container are provided in the JAR manifest according to java standard specifications. Micro-containers JAR files can be run over a classical standard Java Virtual Machine (JVM).

Once running, a micro-container is listening and waiting for hosting service execution requests through the designated port in the deployment descriptor. The packaged service into a running micro-container can be then invoked through our generic client. To that end, we setup the client before invocation by updating its *.properties* file with the appropriate parameters (i.e. micro-containers location, micro-container listening port, operation name to invoke and parameters.).

We also implemented variants of the packaging framework to support migration facilities according to the architectures presented in Section 4.4. The first variant allows building mobile micro-containers using JADE-based migration. To perform this variant, we integrated the JADE-based framework [7] to our packaging process in order to generate mobile micro-containers (See Figure 4.2). This intermediary step allows integrating additional migration technical modules to micro-containers (e.g. context management module). We have also implemented an administrator agent container to deploy with the mobile containers. This latter is responsible of sending the migration requests to the micro-containers.

The second variant of the packaging framework with migration facilities consists on adding a migration facilities package implementing generic migration modules (See Section 4.4.2). The generic migration modules allow adding a migration component in the micro-containers that interact with a receiver container to perform migration (See Figure 4.3).

6.2.3 COAPS API

COAPS API is an OCCI-compliant solution implementing the platform and application resources description and provisioning model that we have defined [89] (See Section 5.2). COAPS API handles *Application* and *Environment* resources. *Environment* resources are in turn composed of platform resources (e.g. *Container*, *Router*, *DatabaseLink*, etc.). COAPS sources are composed of 2 packages, 2 classes and 312 instructions. COAPS API is based on both Representational State Transfer (REST) architecture [90] and the OCCI HTTP Rendering [54]. OCCI HTTP Rendering defines how to interact with an OCCI model to manage its resources while REST architecture describes a style for building distributed systems. REST architectural style is based on resources associated to unique identifiers (e.g. URI). The interactions with these

[6] one-jar.sourceforge.net/
[7] http://jade.tilab.com/

resources are based on a standardized communication protocol (e.g. HTTP).

COAPS API exposes a set of generic and RESTful HTTP operations (i.e. GET, POST, PUT and DELETE) for Cloud applications management and provisioning (See Section 5.3). Concretely, these operations implement actions that can be applied to *Environment* and *Application* resources according to our defined model. We implemented these resources based on the OCCI4java[8] project. OCCI4java implements the OCCI core specifications and the OCCI infrastructure extension. We extended this project to include implementations of our platform and application extensions.

We also developed a generic Web client (see Figure 6.5) for applications provisioning and management in existing PaaS. The Web client acts as an access point for the API implementations and allows the user to call the Environment/Application management operations. Through this client, we show that our API allows a seamless (i.e. using (1) the **same** application/environment manifests structure and (2) the **same** management actions) PaaS application provisioning.

In the following, we discussed two realistic use cases of COAPS API before detailing its current available implementations.

6.2.3.1 COAPS realistic use cases

COAPS API was taken up in the French FUI CompatibleOne[9] project and the European Easi-Clouds[10] project.

CompatibleOne project is an open source project which provides, among others, a platform (i.e. ACCORDS) for Cloud services description and allows interoperability and portability between different Cloud resources provisioned by heterogeneous Cloud environments [91]. The ACCORDS platform authorizes application developers to choose the runtimes and frameworks of their choice to deploy their applications. The developers are not supposed to consider proprietary characteristics related to a specific PaaS. To ensure these requirements, COAPS API and its related resources description model were proposed [92]. Indeed, describing applications using a generic model and using a unified API for their provisioning enabled us to meet the portability challenges across several PaaS.

As a proof of concept for the COAPS module, the demonstration illustrated in Figure 6.3 was presented at the project's final review. For those demonstration, the XWiki company, a project partner, provided us with a Cloud-aware version of their XWiki Enterprise application[11] that we have provisioned in two different PaaS instances (i.e. a Cloud Foundry instance and an Openshift instance) using the same application descriptor and the same actions. XWiki Enterprise application is a light and powerful development platform that allows users to customize the wiki to their specific needs (e.g. sharing documents, monitoring project progress, etc.). This demonstration

[8] https://github.com/occi4java/occi4java
[9] compatibleone.org/
[10] easi-clouds.eu/
[11] http://enterprise.xwiki.org/xwiki/bin/view/Main/WebHome

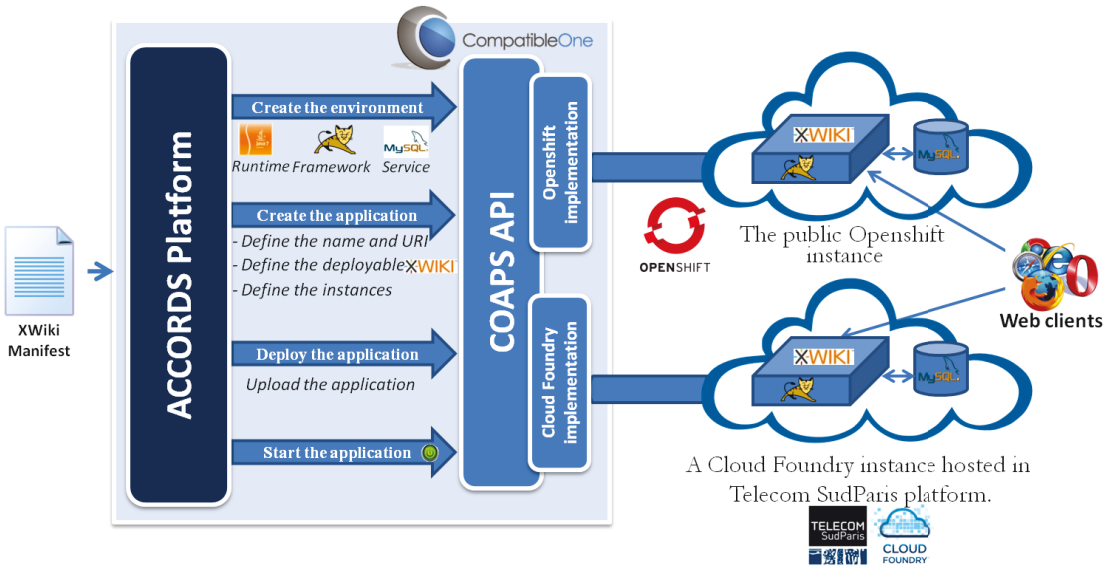


Figure 6.3: COAPS demo presented in the CompatibleOne final review.

shows how the portability of the XWiki Enterprise application is ensured through our solution.

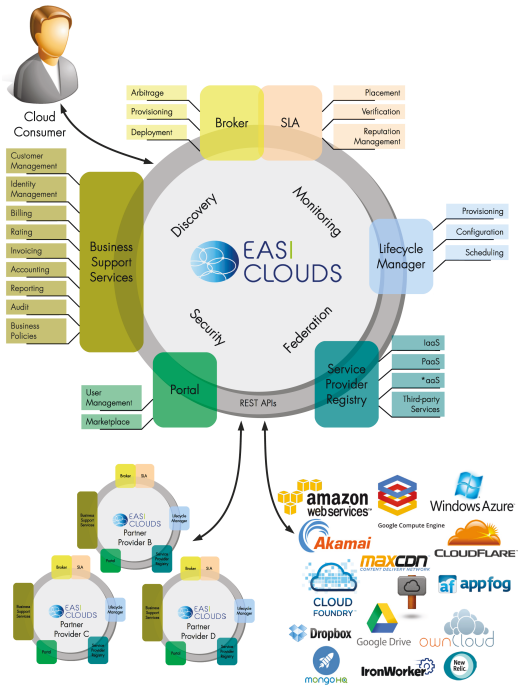


Figure 6.4: Conceptual diagram of the EASI-CLOUDS platform [10].

EASI-CLOUDS project stands for Extendable Architecture and Service Infrastructure for Cloud-Aware Software. This project aims at offering novel and beneficial solutions for both Cloud consumers and providers. The major expected outcome is an open-source Cloud platform, the EASI-CLOUDS platform (see Figure 6.4), *"that can be instantiated to set up an application type-specific cloud (e.g. e-learning, HPC-on-demand, storage marketplace) for a private, public, or hybrid usage, and implementing a given level of security, privacy and QoS"* [93]. An EASI-CLOUDS platform must also provide the required facilities for intra-cloud cooperation and federation.

In this project, one of our objectives is to provide the required facilities promoting EASI-CLOUDS platforms federation. COAPS is used in this context to enable application provisioning/management (i) between EASI-CLOUDS platforms of a same federation and also (ii) with other existing commercial PaaS (see Figure 6.4).

6.2.3.2 Examples of existing COAPS implementations

Currently, we provide a Cloud Foundry and OpenShift implementations (respectively called CF-PaaS API and OS-PaaS API) [89]. These implementations are developed in Java and provided as RESTful Web applications (i.e. WAR). We also developed a generic Web client for application provisioning and management in PaaS with an implementation of our API (See Figure 6.5).

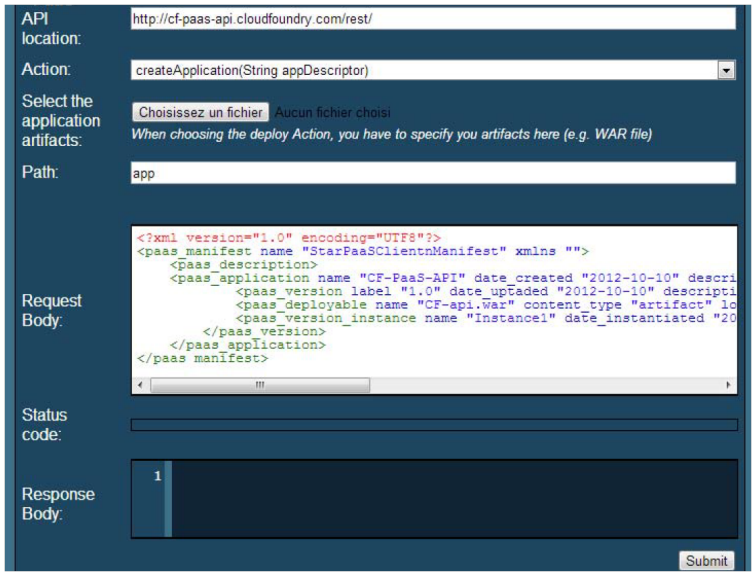


Figure 6.5: CF-PaaS Proxy generic Web client.

Source archives and a demonstration video of application provisioning in Cloud Foundry through CF-PaaS API are available online at [86]. The demonstration video highlights that the resources provisioning and the application provisioning scenario are processed in a reasonable time.

Finnish project partner (i.e. Tampere University) on Easi-Clouds project, which have developed a collaborative IDE for the Cloud called CoRED [94], uses CF-PaaS API to deploy their developed services in Cloud Foundry PaaS. In the near future, they plan to add a new implementation to deploy a part of their services in AppScale PaaS. Moreover, an egyptian project partner (i.e. Cairo University) have implemented a GAE-PaaS implementation to deploy their services in Google App Engine PaaS [95]. All these implementations demonstrate the easy way of developing a new COAPS implementation through the proxy system that we have defined, even by a tierce party, with only the sources and documentation that we provide.

6.3 Experimentations

In this Section, we present and comment the experiments that we have conducted. The first series of experiments was carried out to highlight classical service containers (e.g. Apache Axis, Apache Tomcat, etc.) limitations in Cloud context. The results and ascertainments of these experiments are presented in Section 6.3.1. The second series of experiments was performed to evaluate our service micro-container performances against classical service containers in Cloud environments. The results of these experiments are detailed in Section 6.3.2. Finally, we conducted experiments to evaluate mobile micro-containers performances and migration overhead. The results of these experiments are detailed in Section 6.3.3.

6.3.1 Service containers limitations in Cloud environments

Generally speaking, to deploy an application on a service container, one must mainly provide two elements:

- The application with all its components (e.g. compiled classes, resources, etc.),
- A deployment descriptor that specifies the container options to run the application.

There are several types of service containers. For example, for the J2EE technology there are: Web containers for servlets and JSP, EJB containers for EJBs, and client containers for applications on standalone terminals using J2EE components.

In line with the definition given in [71], we can define a Web container as an application that implements the communication contract between different application components obeying to a distributed architecture. This contract specifies a runtime environment for Web components including safety and competition management, lifecycle, transactions, deployment and other services. Web containers can generally use their own Web server and also be used as a plug-in in a dedicated Web server (as is the case with Apache servers or Microsoft IIS). Examples of Web containers are Tomcat and Axis which are open sources projects from Apache foundation.

As part of our work, we decided to conduct a set of experimentation scenarios involving classical service containers in realistic Cloud context. To that end, we considered Apache Axis 2[12] which is one of the most adopted service containers in the industry world. Therefore, Apache Axis 2 can handle a big number of services at the same time and response to client's queries in an acceptable time. These experimentations allowed us to highlight its lacunas for deployment and management of a huge number of Web services in Cloud environments [58] [59].

Figure 6.6 and Figure 6.7 show respectively the behavior of the response time and memory resources consumption for Apache Axis 2 server regarding the number of deployed services. To perform these experiments, we have used one virtual machine with 0.5 GHz of CPU and 512 Mb of memory. We have also developed a test collection generator to obtain thousands of generated Java Web services code archives and their correspondent WSDL files. The functionality which implements these Web services is basic: calculation of an arithmetic operation of two integers.

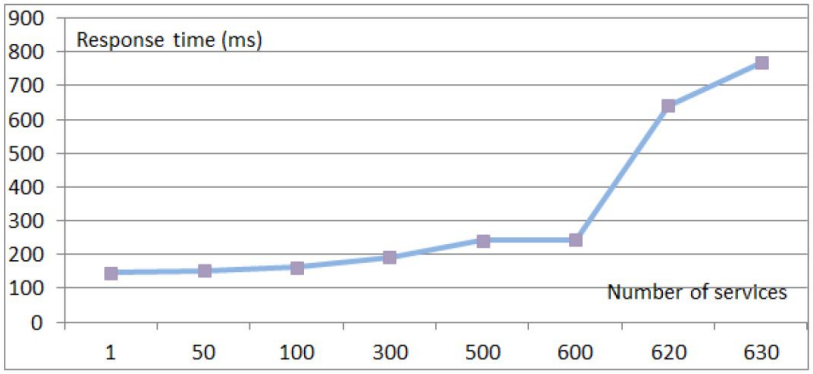


Figure 6.6: Apache Axis 2 server response time evolution.

At each iteration of the experiments, we deployed a number of services set in advance and then we took the measures using a java client. On the one hand, based on the curve shown in Figure 6.6, we note that the response time of an Axis 2 client request is too large from 600 deployed services. We also noted a total crash of Axis 2 from 630 deployed services. On the other hand, based on the curve shown in Figure 6.7, we noticed an important increase of the host machine memory consumption especially for high numbers of services. These two aspects of Axis 2 behavior are characteristics of several classical Web containers we studied and represent the two major defects which prevent these containers to scale and thus makes it unsuitable for Cloud context.

Based on these ascertainments and after studying different architectures of service containers (e.g. See [96] for Apache Axis 2 container, See [97] for Apache Tomcat container, See [98] for Web Services Container Reference Architecture), we realized that they aren't able to scale among many physical machines. Any of those containers

[12]http://axis.apache.org/axis2/java/core/

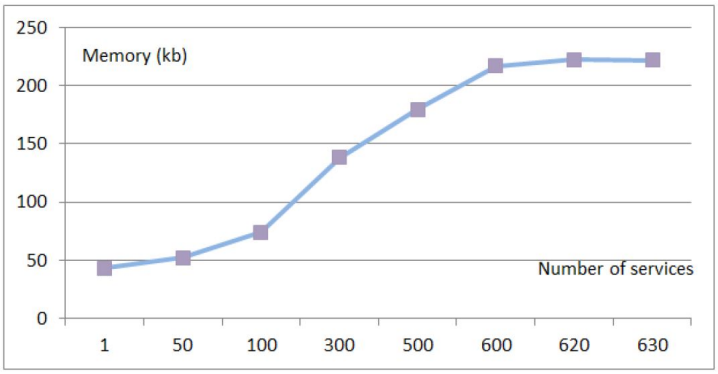


Figure 6.7: Apache Axis 2 server memory consumption evolution.

can be deployed physically just on one machine, so the Cloud using such containers will reach its limits when this host machine uses its entire resources even if the other machines are charge free. Based on this, we can say that the Cloud's limit is the same limit of the host machine in which we deployed the service container. This machine presents then a bottleneck in every Cloud using such containers for service-based applications.

This motivated our decision to design and implement the service micro-containers. We think that if each service is deployed separately, the Cloud can really contain as much deployed services as it is allowed by the available physical resources of the Cloud. Each service can be deployed in a micro-container anywhere in the Cloud with the minimal use of its resources. We can deploy as many micro-containers as it is possible on any machine, if this machine reaches its limit we can deploy on a second one then on a third and so on. With this idea we are sure that we use the minimal resources to encourage the pay as-you-go model of Cloud Computing [99] and we can enforce the elasticity of Cloud because we just use the resources needed.

6.3.2 Service micro-containers experimentations

To perform these experiments, we chose to evaluate the performance of our micro-containers opposite to Apache Axis 2. As far as we know, Axis 2 is one of the most used and efficient classical services containers. These experiments aimed to validate the good behaviour of our micro-containers for a huge number of deployed services in a Cloud context and to demonstrate its superior performances and scalability comparing to classical service containers such as Axis 2.

Firstly, we developed a test collection generator to obtain thousands of generated Web services code archives and their WSDL files. The functionality which implements these Web services is the same: calculation of an arithmetic operation of two integers. At each iteration of the experiments, we deployed a number of services set in advance, we invoke one of these services randomly selected using a classic java client and then

we took the measures.

We have considered a couple of criteria that we think essential to evaluate the two service containers 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.

The experiments was conducted in the Network and Cloud Federation (NCF) experimental platform deployed at Télécom SudParis. When we conducted these experiments, the NCF platform has: 380 Intel Xeon Nehalem Cores, 1.17 TB RAM and 100 TB as shared storage. We have used OpenNebula[13] resources manager to create our experimental VMs using various personalized templates. Characteristics of the templates we used for experimentations are detailed in Table 6.3.

Table 6.3: Used templates details for VMs instantiation.

Template name	**CPU (MHz)**	**Memory (kb)**
T1	1	1024
T2	1	512
T3	0.5	1024
T4	0.5	512
T5	0.25	512
T6	0.25	128
T7	0.25	64
Tp	3	4096

To perform these tests, we defined several scenarios with different alternatives. These scenarios reflect the objectives that we want to highlight in our experiments. The details of these experiments are as follows:

1. Compare service containers performance in different VMs,
2. Confront Axis 2 container and the micro-container (MC) in VMs with low memory,

[13] opennebula.org

3. Confront Axis 2 container and MC in VMs with low CPU power,
4. Confront Axis 2 container and MC in overpowering VMs,
5. Compare the number of used VMs by Axis 2 and micro-container to deploy the same huge number of services.

Obtained results of each one of these scenarios are detailed in the rest of this Section.

Axis 2 Versus Micro-container with various VMs

In the first series of tests, we deployed just one service on Axis 2 and on the micro-container. After that, we deployed these containers on different virtual machines created by various templates listed in Table 6.3 and then we took measurements. The purpose of this experiment is to see the impact of the VM template choice on performance of the two containers. Figure 6.8 shows the different stored values for Axis 2 and MC for one service response time deployed in these VMs while Figure 6.9 shows the evolution of their memory resources consumption.

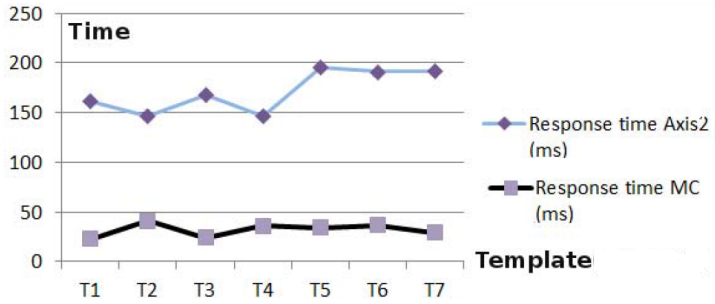


Figure 6.8: Response time evolution with different VMs templates (Axis 2 Vs MC).

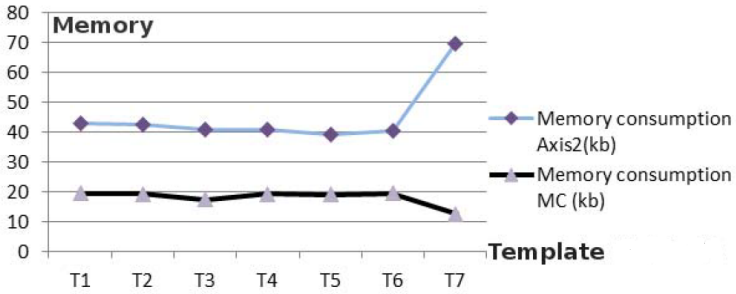


Figure 6.9: Memory consumption evolution with different VMs templates (Axis 2 Vs MC).

These experiments show that for the same deployed service and whatever is the template used to instantiate the hosting VM, the micro-container performances are

better than Axis 2. The micro-container sends back responses faster than Axis 2 and consumes less memory. However, for both service containers, we do not notice any major changes in performance when changing the VMs template except when using T7 template. Indeed, for this template increases for Axis 2 server and decreases for service micro-containers.

Axis 2 Versus Micro-container with less memory VM (T4 template)

For this experimentation scenario, we have chosen to use identical VMs to host the two containers and vary the number of deployed services. The template we used to instantiate VMs is T4, a low memory template. Figure 6.10 shows the different stored values for response time experiments.

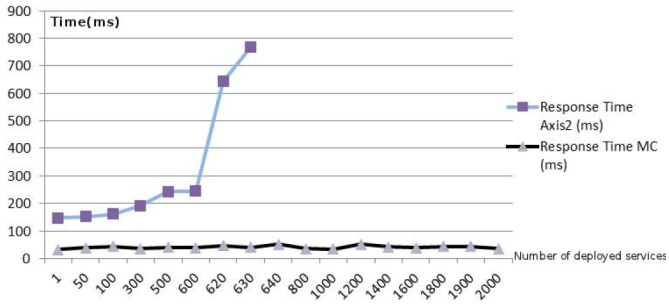


Figure 6.10: Response time evolution-Axis 2 Vs MC (T4 VM template).

During these experiments, we had to make a choice between:

- Alternative 1: Test by comparing Axis 2 performance versus a single instance of the micro-container performance,
- Alternative 2: Test by comparing total CPU time between all instances of deployed micro-containers running in parallel versus Axis 2.

Finally, we opted for the first alternative plan because we chose to compare performance of the two service containers with the same test collection of deployed services.

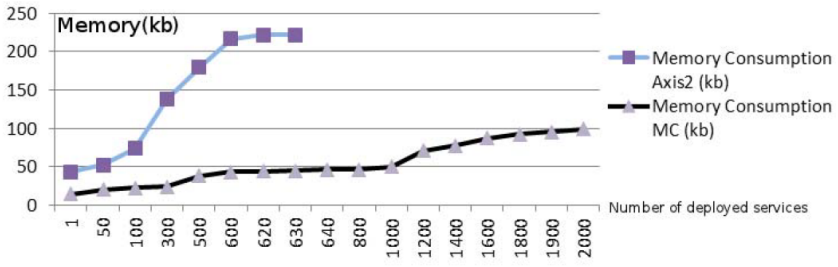


Figure 6.11: Memory consumption evolution-Axis 2 Vs MC (T4 VM template).

Based on the curves schematized in Figure 6.10, we note that Axis 2 response time increases proportionally to the number of deployed services. Concretely, a major part of this time evolution lies on the time needed for Axis 2 to update hosted services indexation mechanisms, manage execution contexts and load the requested service. Processing service request, service execution and response building process times are roughly steady. For our micro-container, the response time is approximately the same for all the experiences. This can be explained by the fact that each instance of the micro-container is independent from the others, and hosts only one service (no service context management then). Hence, we can deploy as many micro-containers as it is possible regarding the available resources in the virtual machine without affecting the response time. In this testing environment, Axis 2 crashed when 630 services are deployed because there is no more memory resources on the hosting virtual machine for deploying and processing more services. Actually, Axis 2 overflowed into the virtual machine due to its excessive memory resources consumption. However, our micro-container reached more than 2000 deployed services using our defined approach without any performance degradation. These interpretations are also verified by the memory consumption measures presented in Figure 6.11.

Axis 2 Versus Micro-container with less CPU VMs (T5 template)

In this experimentation scenario, we repeated the same tests with changing only the template used to intantiate the hosting VMs. We used T5 template in order to increase the memory capacity of the hosts (1Gb of memory instead of 512 Mb) and low CPU power. The purpose of this operation is to avoid Axis 2 crash observed in the previous experiment by providing more memory in the hosting VMs. Figure 6.12 shows the different stored values for response time experiments while Figure 6.13 shows memory resources consumption evolution for the two containers.

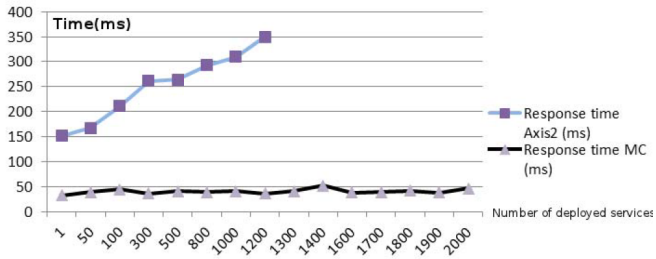


Figure 6.12: Response time evolution- Axis 2 Vs MC (T5 VM template).

We notice that the memory usage is linear and increases according to the evolution of the number of deployed services in the two containers. Obtained curves show the savings of the micro-container against Axis 2 in memory usage. This is due to the large number of programs and operating files generated by the Axis 2 core to index and manage hosted services (e.g. archives, index, temporary files, context files, etc.).

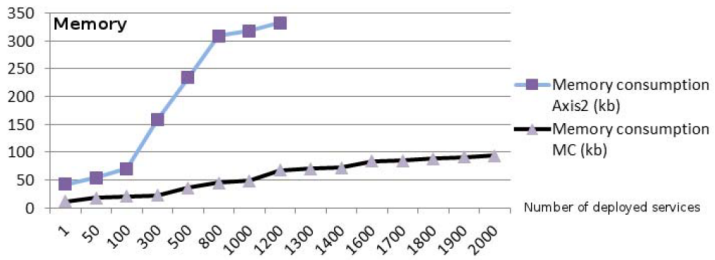


Figure 6.13: Memory consumption evolution-Axis 2 Vs MC (T3 VM template).

Axis 2 has exceeded its limit observed in the latest experience due to the lack of memory, but it still crashing when we increase the number of deployed services (1230 deployed services for this test). This crash is due to an overflow of the hosting VM CPU. On the other side, for the same VMs, we deployed more than 2000 services on micro-containers with steady response times. All this inspired us to repeat these tests with overpowering templates. The detail of these tests is detailed in the next subsection.

Axis 2 Versus Micro-container with powerful VM (Tp template)

In this experimentation scenario, we create powerful VMs instantiated using Tp template in order to eliminate all physical limit aspects which penalized Axis 2 during last scenarios. Figure 6.14 schematizes the obtained response time curve, while Figure 6.15 schematizes the obtained memory consumption curve.

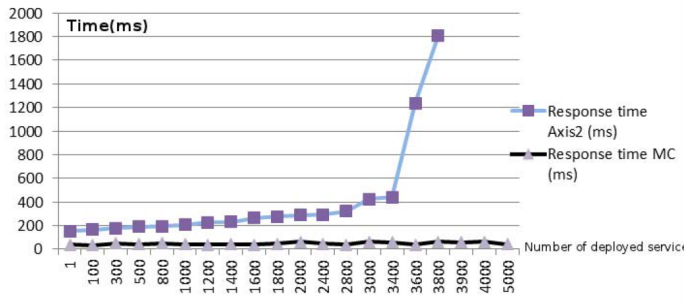


Figure 6.14: Time response evolution-Axis 2 Vs MC (Tp VM template).

In this experiments, we note that Axis 2 reaches and exceeds the limits observed in previous scenarios. However, it crashes for 3860 deployed services while CPU and memory resources are still available in the hosting VM. In fact, this crash is caused by the design of Axis 2 itself, which begins to fail from a given number of services even if the host still have resources. This is an intrinsic limitation related to the architecture of Axis 2, which is certainly efficient and adequate for an industrial use but still

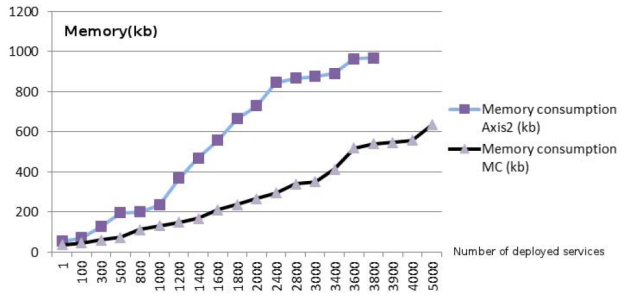


Figure 6.15: Memory consumption evolution-Axis 2 Vs MC (Tp VM template).

unreliable for such environments. This is true for all classical service containers (e.g. Apache Tomcat, Metro, GlassFish, etc.).

In the same testing environment, we was able to deploy more than 4000 services in micro-containers with steady response time and memory consumption values. Indeed, when using micro-containers, the limit number of deployed services is the physical limit of the VM while Axis 2 crashes when it reaches its logic limit of supported hosting services.

Based on these curves, we observed that just before the Axis 2 crash, response times increase abruptly. In fact, for a huge number of services, Axis 2 has to manage much indexes and service contexts which led it to consume more resources and degrades its performance. The idea that we had was to determine the optimal range number of services that Axis 2 can host and then confront the two containers performance in this range. This can easily be determined by a simple calculation of the memory space ratio used for a number of deployed services in Axis 2 based on the values of the curve in Figure 6.15 (i.e. between 800 and 1000 services for these experiments). Recorded micro-container performance values for this range are better than the Axis 2 values.

Axis 2 Versus Micro-container with mutiple VMs usage (T6 template)

In this experimentation scenario, we use multiple Axis 2 instances deployed on multiple VMs. We deploy services respectively on Axis 2 and MC until the saturation of the hosting VM. Then, we instantiate a second with a second Axis 2 instance, and we resume deployment until the saturation of the second host and so on. This scenario simulates horizontal scalability in a Cloud provider. For these tests, we used T6 VM template to reach quickly the VMs limits. When multiple VMs are used, the memory consumption values represent the sum of all memory spaces used in all allocated VMs.

The curves related to these experiments are shown in Figure 6.16 and Figure 6.17. Using Axis 2, we used 4 VMs to host 1200 services while we used only 1 VM when using micro-containers. For the same number of used VMs (i.e. 4 VMs) we have successfully deployed more than 5000 services using micro-containers, while we could deployed only 1200 services using Axis 2. Based on this, we can say that not only that

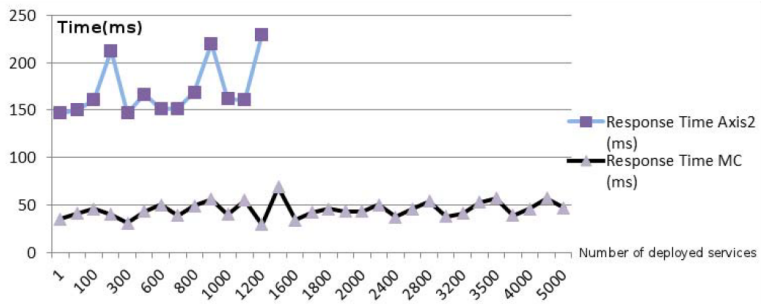


Figure 6.16: Time response evolution in multiple VMs-Axis 2 Vs MC (T6 VM template).

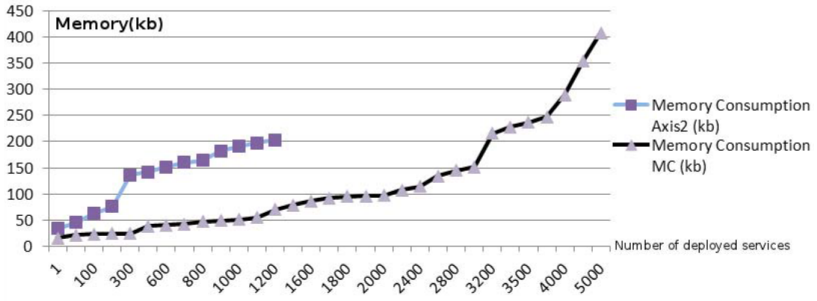


Figure 6.17: Memory consumption evolution with multiple VMs-Axis 2 Vs MC (T6 VM template).

micro-container performance are superior to Axis 2 performance but also that using micro-container on Cloud providers costs less than using classical service containers.

6.3.3 Mobile service micro-container experimentations

We also conduct a set of experiments to evaluate the mobile micro-containers performances and determine the overhead of the added migration. We decided to evaluate our two variants of mobile micro-containers against classical micro-container to evaluate their performances when they provide the same functionality (i.e. migration). To do this, we have considered the same test collections of services (i.e. calculation of an arithmetic operation of two integers provided as inputs), the Tp template to instantiate hosting VMs and the same comparison criteria (i.e. server response time and memory resources consumption) used to evaluate micro-container against Axis 2. The results of the experimentation of the two variants of the mobile micro-containers are detailed in the rest of this Section.

JADE-based service micro-container experimentations

Figure 6.18 shows behaviour of obtained response time values for both JADE-based micro-container and classical micro-container when we vary the number of deployed

services while Figure 6.19 shows behaviour of memory resources consumption values. Response time values of both containers increase when we increase the number of deployed services. Response time values are approximately similar with a slight overhead for the JADE-based migration. We note also that the JADE platform crashes when deploying more than 2500 services even though there are available memory and computing resources in the hosting VM.

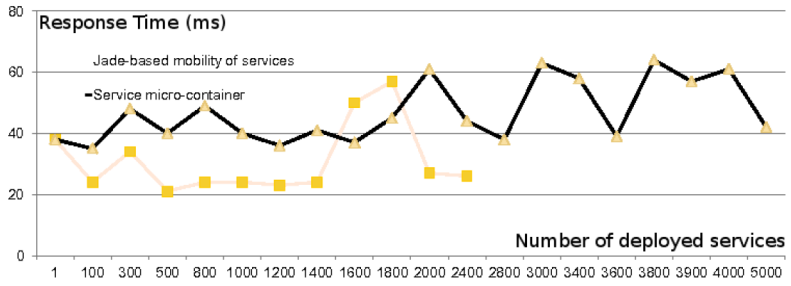


Figure 6.18: Response time curve (JADE-based migration Vs MC).

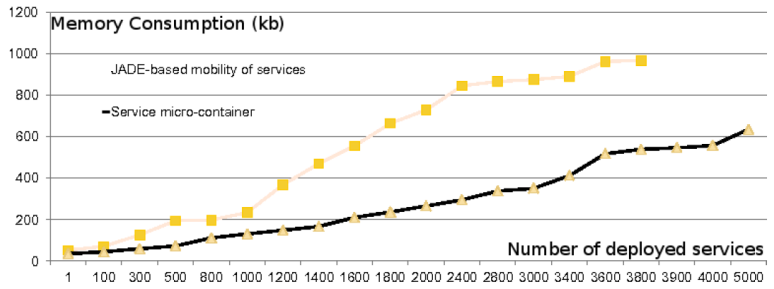


Figure 6.19: Memory consumption curve (JADE-based migration Vs MC).

For memory consumption experiments, we quantify that the used memory overhead for JADE-based micro-containers exceeds 50% regarding the classical micro-container memory consumption. Based on these measurements, we can conclude that adding migration to service micro-containers using JADE platform does not affect service micro-container response time. However, JADE platform usage engenders an important memory consumption overhead and it crashes when a huge number of deployed services is reached. These limits are due to the architecture of JADE multi-agent platform which is composed of agents' server that consume a lot of memory resources. This Agent server is a technical component of the JADE platform and mandatory for its deployment and execution. It is responsible of managing the life cycle of the mobile agents and maintaining an up-to-date list of agents with their location.

Migration-based service micro-container experimentations

Figure 6.20 shows behaviour of response time values for both mobile micro-container and classical micro-container. The two curves shape is almost identical except for a few minor differences in some measures. This can be explained by our approach when we implemented the migration module. In fact, this module is implemented and integrated in classical micro-containers as an extension; we have not modified the architecture and the basic modules (i.e. communication module, invocation module) of the micro-container.

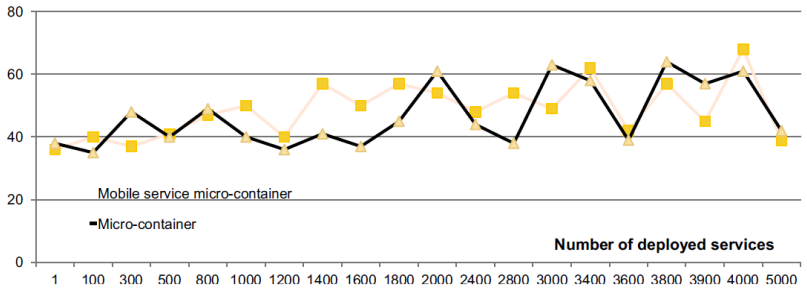


Figure 6.20: Response time curve (M-MC Vs MC).

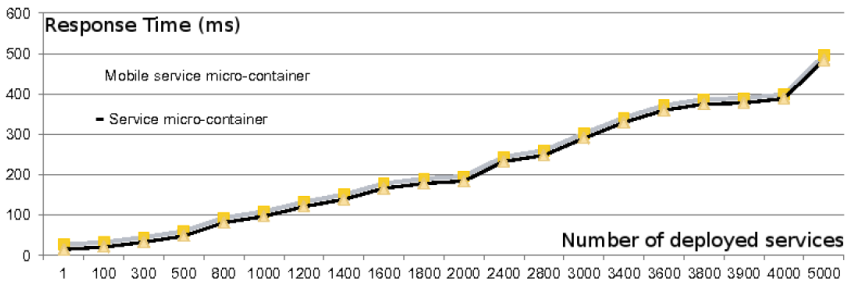


Figure 6.21: Memory consumption curve (M-MC Vs MC).

Figure 6.21 shows behaviour of the memory resources consumption for both containers. The curves shape is identical with a light overhead for mobile micro-container. This overhead is due to memory resources consumed by the receiver but remains nevertheless neglected. Indeed, we need only one receiver per VM to manage all the micro-containers hosted in this VM. We conclude the migration overhead in this case is not significant compared to JADE-based micro container overhead.

6.4 Use case: Provisioning of autonomic applications

In the following, we present a realistic use case resuming our findings. This use case consists in provisioning applications with autonomic computing capacities in Cloud platforms. Autonomic applications are applications able to monitor and reconfigure their components autonomously based on specific technical components to be deployed with the application. We present in the first part of the Section the context and the purpose of autonomic applications provisioning and management in Cloud platforms. Then, we detail the implementations that we have performed to acheive the stated objective in the second part of the Section.

6.4.1 Context and purpose of the use case

According to studies that we carried, Cloud platforms do not support monitoring and reconfiguration for deployed applications together at the same time. Indeed, a human intervention is still required to interpret, as it should be, the application monitoring data and act based on these data in order to maintain the required execution of the application. Therefore, Cloud providers do not support provisioning of autonomic applications. These applications are able to manage, automatically and dynamically, their required resources to respect their Service Level Agreement (SLA). In this context, we propose a novel approach to provision autonomic applications in existing Cloud platforms.

To provision an autonomic application in a Cloud platform, we couple:

1. An OCCI-based autonomic resources description model and an API implementing this model [77],
2. Our OCCI-based platform and application resources description model and COAPS as its implementation.

6.4.2 Implementation and validation

Concretely, our approach proposes to dynamically add autonomic management facilities to applications when deploying them in a target PaaS. As for elastic SBPs provisioning approach, our novel framework requires no modification on the Cloud system side and can be deployed and supported by any PaaS thanks to our performed generic provisioning mechanisms.

The performed system is schematized in Figure 6.22. To establish our autonomic computing framework, we start by setting up an OCCI Server. This server encompass COAPS as the PaaS interface, implements both autonomic resources and platform/application description models and is responsible of instantiating and managing OCCI resources.

The first Resource instantiated in this server is the *Autonomic Manager* Resource. The *Autonomic Manager* is responsible of preparing the *Application* resource based

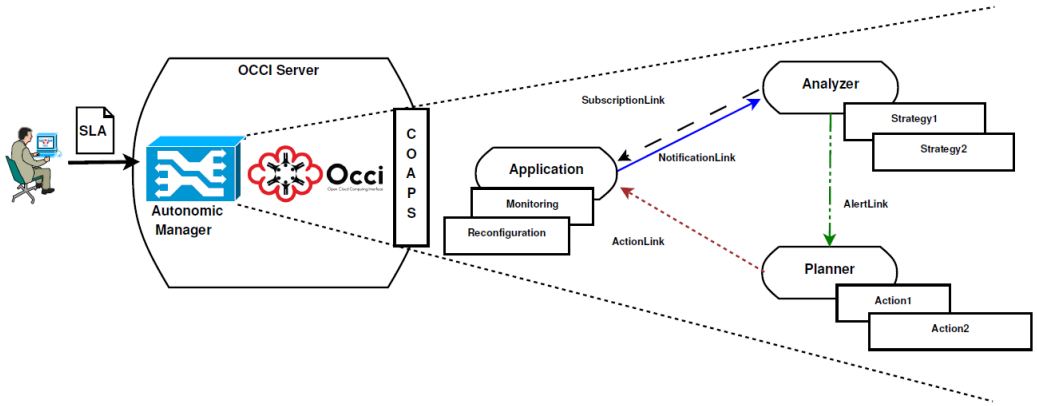


Figure 6.22: Autonomic Computing framework overview.

on the provided SLA before its deployment. To perform that, the *Autonomic Manager* detects the attributes and/or services that need to be monitored for the application. Then, it extends the application artifacts by adding necessary monitoring Mixins (i.e. Polling and/or Subscription) and a *Reconfiguration* Mixin to enable reconfiguration facilities. After that, the *Autonomic Manager* sends a request to COAPS in order to instantiate resulting *Application* Resource (i.e., the basic Application Resource with their newly added Mixins), deploy it in the target PaaS and start it. We create firstly the hosting *Environment* resource. Then, we create the *Application* resource and we process deployment by associating the given *EnvId* to the *AppId* before starting the *Application*.

After that, the *Autonomic Manager* instantiates and customizes the needed Resources and Links in order to establish the autonomic framework. For ease of presentation, we refrain from presenting all the Mixins in Figure 6.22. Therefore, we kept just the needed Mixins instances of *Strategies* Mixin (i.e. Strategy1 and Strategy2) and *Reconfiguration Actions* Mixin (i.e. Action1, Action2). The *Autonomic Manager* instantiates the *Analyzer* and subscribes it to the *Application* Resource. The *Analyzer* may receive notifications through an instance of the *Notification* Link. At the reception of a notification, the *Analyzer* uses *Strategies* Mixin to process incoming monitoring information. If one of the strategies is verified, the *Analyzer* may raise alerts to the *Planner*. Accordingly, the *Autonomic Manager* instantiates the *Planner* and links it to the *Analyzer* using an *Alert* Link. The *Planner* generates then a plan for reconfiguration actions. The used plans are responsible of generating reconfiguration actions.

The last step is to link the *Planner* to the *Application* Resource using an *Action* Link that can use the generated reconfiguration actions and applies them.

This work allows us to provision already performed autonomic computing framework and autonomic applications in Cloud platforms that do not provide basically

monitoring and reconfiguration facilities. In addition to that, the use of COAPS allows us to make this provisioning PaaS-independent.

6.5 Conclusion

In this Chapter, we presented the implementations that we have done to validate our SPD approach and prove its feasability and good performance. The total metrics of all developed tools are 45 packages, 287 classes and 60325 instructions.

For the first step of our approach, we achieved *BPEL2Java* and *SCA2java* tools that implements our defined slicing and aggregation algorithms for both BPEL-based processes and SCA-based applications. After that, for the second step, we provide details about the implementation procedure and technologies that we have used to implement the packaging framework and service micro-containers building. For the third step, we presented COAPS API that we have developed to implement our platform and application resources description model and to perform deployment in Cloud platforms. Results and interpretation of service micro-containers performance experimentations against Apache Axis 2 server in Cloud environment are also provided.

In the last part of the Chapter, we presented a realistic use case consisting in provisioning autonomic applications in Cloud platforms using COAPS API.

Chapter 7

Conclusion and Perspectives

7.1 Conclusion

Service-based applications are described according to Service Oriented Architecture (SOA) and consist of assembling a set of elementary and heterogeneous services using appropriate service composition specifications such as Service Component Architecture (SCA), Business Process Model and Notation (BPMN) or Business Process Execution Language (BPEL). These applications are built from components and services that may be heterogeneous in the sense that they (1) are not all implemented using the same programming languages (e.g. C++, Java, etc.), (2) do not support all the same communication protocols (e.g. RMI, SOAP/HTTP, etc.) and/or (3) do not run on the same hosting frameworks (e.g. POJO VM, .NET framework, etc.).

Provisioning a service-based application in the Cloud consists of: (1) allocation of adequate resources to host and execute the application and (2) upload of the application artifacts (e.g. binary code) on the allocated resources. This provisioning task requires then the delivery of appropriate frameworks and specific runtimes supporting the heterogeneity of the application components. In addition to that, such applications are often distributed and require occasionally deploying their components separately on multiple Cloud platforms. Meanwhile, existing Cloud platforms has proprietary description models to describe, manage and provision applications and their hosting resources. They expose also proprietary and heterogeneous user APIs (e.g. proprietary operations, specific provisioning scenarios, etc.).

To tackle this issue, we defined in this thesis an approach that we called SPD to provision service-based applications in Cloud environments. The SPD approach consists in three steps:

1. Slicing the application into a set of elementary services,
2. Packaging the resulted services into service micro-containers,
3. Deploying the micro-containers in a target Cloud environment.

Since service-based applications have heterogeneous components it is difficult to satisfy its requirements by provisioning one allocated hosting environment based on existing Cloud environment capabilities. Therefore, we propose to slice the applications into a set of elementary and autonomous services before allocating a dedicated and appropriate environment for each one of the obtained services. In this step, we considered: (i) applications modeled as business processes and can be formally represented using Petri nets and (ii) applications modeled as composition of service components that can be represented using graph-based composition. The use of formal representations allows us to preserve semantics of sliced applications. For applications modeled as business processes, we defined algorithms to slice their correspondent Petri net into a set of dependent WF-nets and to determine the choreography schema to follow for their execution. We also provided the proof of preservationof initial business process business semantics when executing the WF-nets. We provide an implementation (i.e. BPEL2Java) and an illustrative example supporting the slicing of a BPEL process and the aggregation of its obtained services according to transformation rules that we have defined. For applications modeled as composition of service components, we defined an algorithm to slice its correspondent directed graph into a set of elementary services and aggregate them. We provide an implementation of this algorithm (i.e. SCA2Java) and an illustrative example supporting slicing SCA-based applications.

For the second step of our approach, we define a packaging framework architecture that allows service micro-containers building around a given service from generic modules. Only necessary resources to implement service binding types, such as communication protocols, are selected from the packaging framework and encapsulated in the generated micro-container to host the service. We also propose extensions of this framework to include support of a set of non functional properties, such as migration, monitoring and elasticity, if they are required by the developer. We provide implementation of the packaging framework for Java services communicating in HTTP/SOAP. We conduct several experimentations to (1) highlight the classical service containers limitations in Cloud environments, (2) show the superior performance of our micro-containers against these service containers and (3) determine the overhead of the migration facilities if they are included to a micro-container.

In the third step of our approach, we can deploy the service micro-containers in existing Cloud infrastructures (IaaS) and Cloud platforms (PaaS). The deployment in IaaS consists in instantiating VMs and uploads the service micro-containers on them before starting their execution as standalone applications. For the deployment in a PaaS, we defined a generic description model for applications and PaaS resources. Our proposed description model is based on the OCCI standard. According to OCCI, each identified PaaS resource is characterized by a set of attributes, management actions and associate lifecycle. Based on this model, we are able to describe, provision, and manage applications and PaaS resource in a unified way whatever is the target Cloud platform. We also developed a PaaS-independent REST API called COAPS implementing this model to process applications deployment and management on tar-

get PaaS. We provide COAPS specifications proxy architecture and current available implementations (e.g. CF-PaaS API, OS-PaaS API, etc.). We comment and discuss the deployment of the micro-containers stemming from our illustrative examples. The deployment in IaaS is performed on Network and Cloud Federation (NCF) infrastructure deployed at Télécom SudParis, while the deployment in PaaS is performed in Cloud Foundry through COAPS.

As a realistic use case of our contributions, we present and detail the process of provisioning autonomic applications in Cloud platforms. Autonomic applications are applications able to monitor and reconfigure its components autonomously based on specific technical components to be deployed with the application. Thanks to COAPS, the application and the related autonomic computing framework are provisioned and started in Cloud Foundry.

To conclude, our defined SPD approach provide flexible deployment mechanisms to support the strong heterogeneity of the service-based application components and generic provisioning procedures to allow applications portability and automate and unify the resources allocation and applications deployment whatever is the target Cloud environment. Furthermore, the SPD approach easily integrates with existing applications and handled resources and requires no modification in the Cloud environment side.

7.2 Future work

In the future work, we aim at extending our proposed generic resources description model to include:

1. Cloud platforms management support,
2. Mobile collaborative computing devices and capacities support.

These two perspectives are detailed in the following.

7.2.1 Cloud platform management

Our proposed description model (See Section 6.2) allows the description and the management of applications and generic platform resources. COAPS API which implements this model (See Section 6.3) enables provisioning and handling these resources in existing Cloud platforms.

As perspective of this work, we propose to extend our model and its correspondent API to support management of not only the hosting and execution resources (e.g. service containers, databases, etc.) but also technical platform components (e.g. platform load balancers, monitoring services, etc.).

This extension is indented to platform administrators profile and plans to add new mechanisms for managing efficiently Cloud platforms. In fact, existing PaaS provides

tools and admin APIs allowing administrating their technical resources (e.g. duplicate DEA component in Cloud Foundry, monitor the load balancer in Heroku, refresh Health Controller component in Cloud Foundry, etc.). As it is the case for applications and hosting resources, these resources and correspondent APIs are proprietary and specific per PaaS.

In this Context, we propose to census all technical Cloud platforms resources and define a novel generic model allowing their description and management. We plan also to propose a generic admin API allowing administrating these resources in a unified way whatever in the target PaaS.

7.2.2 Mobile collaborative computing applications and resources provisioning

The growth of mobile applications using Cloud resources for computing and storage (e.g. Instagram, Dropbox, etc.) has induced the proliferation of mobile devices which collaborates each others, usually in real time, by taking advantage of underlying Cloud resources.

Generally speaking, collaborative computing paradigm presents a unified data processing model in which Services, Processes and end users collaboratively work on shared data towards a common goal [100]. Mobile collaborative computing suggests that the end user devices are made for portability, and are therefore both compact and lightweight. Performances of such devices are limited hence the usefulness of using collaborative computing.

The involved Cloud resources in this process are often specific. On one side, the handled applications are designed and developed for collaborative computing or include features to let users work together over networks (e.g. Microsoft Office and Exchange, Lotus Notes, Videoconferencing applications, etc.). On the other side, mobile applications require often specific execution frameworks and runtimes (e.g. Shared Services Framework (SSF), Android application framework, etc.) to meet with the portable devices characteristics. Moreover, the relationships between the system components (i.e. applications, mobile end users and available Cloud resources) are particular. Indeed, the allocated Cloud resources for applications execution and data storage are ad-hoc while the application must be all the time available. Furthermore, the management of the incoming/outgoing resources must be transparent for the user. All this therefore affects the way in which we describe the applications, their execution resources and the way in which we provision them.

In this context, we propose to (1) extend our resources description model to include describing and provisioning support of mobile collaborative applications and resources and (2) extend COAPS API by providing additional operations and descriptors to support provisioning and managing these resources.

Bibliography

[1] Winston.W Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of the 9th International Conference on Software Engineering*, ICSE '87, pages 328–338, Los Alamitos, CA, USA, 1987. IEEE Computer Society Press.

[2] Tom Kirkham and Keith Jeffrey. PaaSage Project. Model Based Cloud Platform Upperware. Initial Architecture Design. Technical report, November 2013.

[3] Dana Petcu, Silviu Panica, Călin Şandru, Ciprian Dorin Crăciun, and Marian Neagul. Experiences in Building an Event-driven and Deployable Platform As a Service. In *Proceedings of the 13th International Conference on Web Information Systems Engineering*, WISE'12, pages 666–672, Berlin, Heidelberg, 2012. Springer-Verlag.

[4] Eleni Kamateri, Nikolaos Loutas, Dimitris Zeginis, James Ahtes, Francesco D'Andria, Stefano Bocconi, Panagiotis Gouvas, Giannis Ledakis, Franco Ravagli, Oleksandr Lobunets, and KonstantinosA. Tarabanis. Cloud4SOA: A Semantic-Interoperability PaaS Solution for Multi-cloud Platform Management and Portability. In Kung-Kiu Lau, Winfried Lamersdorf, and Ernesto Pimentel, editors, *Service-Oriented and Cloud Computing*, volume 8135 of *Lecture Notes in Computer Science*, pages 64–78. Springer Berlin Heidelberg, 2013.

[5] Cloud Federations in Contrail. In *Euro-Par 2011: Parallel Processing Workshops*, volume 7155 of *Lecture Notes in Computer Science*, pages 159–168. Springer Berlin Heidelberg, 2012.

[6] Topology and Orchestration Specification for Cloud Applications. . Technical report, OASIS, 2012.

[7] Jacques Durand, Adrian Otto, Gilbert Pilz, and Tom Rutt. camp - spec - v1.1 - cs prd02 12 February 2014 Standards Track Work Product Copyright © O ASIS Open 2014 . All Rights Reserved. Page 1 of 93 Cloud Application Management for Platforms Version 1.1. Technical report, February 2014.

[8] Ralf Nyren, Andy Edmonds, Alexander Papaspyrou, and Thijs Metsch. Open Cloud Computing Interface - Core. Technical report, 2011.

[9] Thijs Metsch and Andy Edmonds. Open Cloud Computing Interface - Infrastructure. Technical report, 2011.

[10] The Easi-Clouds project presentation poster. http://easi-clouds.eu/wp-content/uploads/2012/11/EASI-CLOUDS_poster_2012_v4.pdf, 2014.

[11] Peter Mell and Timothy Grance. The NIST Definition of Cloud Computing. Technical Report Special Publication 800 - 145, 2011.

[12] Frederick Hirsch, John Kemp, and Jani Ilkka. *Introduction to Service-Oriented Architectures.*

[13] Michael Beisieg, Henning Blohm, Dave Booz, Jean-Jacques Dubray, Mike Edwards, Bill Flood, Bruce Ge, Oisin Hurley, Dan Kearns, Mike Lehmann, Jim Marino, Martin Nally, Greg Pavlik, Michael Rowley, Adi Sakala, Chris Sharp, and Ken Tam. SCA-Service Component Architecture. Technical report, November 2005.

[14] Business Process Model and Notation (BPMN). Technical report, 2011.

[15] OASIS Web Services Business Process Execution Language (WSBPEL). https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsbpel, 2014.

[16] Bhaskar Rimal, Eunmi Choi, and I. Lumb. A Taxonomy and Survey of Cloud Computing Systems. In *INC, IMS and IDC, 2009. NCM '09. Fifth International Joint Conference on*, pages 44–51, Aug 2009.

[17] Deploying Application Server in Cloud Foundry Using the Standalone Framework. http://blog.cloudfoundry.com/2012/06/18/deploying-tomcat-7-using-the-standalone-framework/, 2014.

[18] Stefan Walraven, Eddy Truyen, and Wouter Joosen. Comparing PaaS offerings in light of SaaS development. *Computing*, 96(8):669–724, 2014.

[19] Ian Foster, Yong Zhao, Ian Raicu, and Shiyong Lu. Cloud Computing and Grid Computing 360-Degree Compared. In *Grid Computing Environments Workshop, 2008. GCE '08*, pages 1–10, Nov 2008.

[20] Yefim V.Natis. Service-Oriented Architecture Scenario. Technical report, Gartner, April 2003.

[21] Xiang Fu, Tevfik Bultan, and Jianwen Su. Analysis of Interacting BPEL Web Services. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 621–630, New York, NY, USA, 2004. ACM.

[22] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More.*

[23] Service Component Architecture (SCA). http://www.oasis-opencsa.org/sca, 2014.

[24] Jim Marino and Michael Rowley. *Understanding SCA (Service Component Architecture)*. Addison-Wesley Professional, 1st edition, 2009.

[25] Michael Beisiegel, Nickolas Kavantzas, Ashok Malhotra, Greg Pavlik, and Chris Sharp. SCA Policy Association Framework. In *Proceedings of the 4th International Conference on Service-Oriented Computing*, ICSOC'06, pages 613–623, Berlin, Heidelberg, 2006. Springer-Verlag.

[26] Nicolas Ferry, Alessandro Rossini, Franck Chauvel, Brice Morin, and Arnor Solberg. Towards Model-Driven Provisioning, Deployment, Monitoring, and Adaptation of Multi-cloud Systems. In *Proceedings of the 2013 IEEE Sixth International Conference on Cloud Computing*, CLOUD '13, pages 887–894, Washington, DC, USA, 2013. IEEE Computer Society.

[27] Nicolas Ferry, Franck Chauvel, Alessandro Rossini, Brice Morin, and Arnor Solberg. Managing Multi-cloud Systems with CloudMF. In *Proceedings of the Second Nordic Symposium on Cloud Computing & Internet Technologies*, NordiCloud '13, pages 38–45, New York, NY, USA, 2013. ACM.

[28] Sergio García-Gómez, Miguel Jiménez-Gañán, Yehia Taher, Christof Momm, Frederic Junker, József Bíró, Andreas Menychtas, Vasilios Andrikopoulos, and Steve Strauch. Challenges for the comprehensive management of Cloud Services in a PaaS framework. *Scalable Computing: Practice and Experience*, 13(3), 2012.

[29] Guillaume Pierre and Corina Stratan. ConPaaS: A Platform for Hosting Elastic Cloud Applications. *Internet Computing, IEEE*, 16(5):88–92, Sept 2012.

[30] Paolo Romano, Luis Rodrigues, Nuno Carvalho, and Joäo Cachopo. Cloud-TM: Harnessing the Cloud with Distributed Transactional Memories. *SIGOPS Oper. Syst. Rev.*, 44(2):1–6, April 2010.

[31] Sergio García-Gómez, Miguel Jiménez-Gañán, Yehia Taher, Christof Momm, Frederic Junker, József Bíró, Andreas Menychtas, Vasilios Andrikopoulos, and Steve Strauch. Challenges for the comprehensive management of Cloud Services in a PaaS framework. *Scalable Computing: Practice and Experience*, 13(3), 2012.

[32] B. Rochwerger, D. Breitgand, E. Levy, A. Galis, K. Nagin, I.M. Llorente, R. Montero, Y. Wolfsthal, E. Elmroth, J. Caceres, M. Ben-Yehuda, W. Emmerich, and F. Galan. The Reservoir model and architecture for open federated cloud computing. *IBM Journal of Research and Development*, 53(4):1–11, July 2009.

[33] Jiang Dejun, Guillaume Pierre, and Chi-Hung Chi. Resource Provisioning of Web Applications in Heterogeneous Clouds. In *Proceedings of the 2Nd USENIX Conference on Web Application Development*, WebApps'11, pages 5–5, Berkeley, CA, USA, 2011. USENIX Association.

[34] Gunho Lee, Byung-Gon Chun, and H. Katz. Heterogeneity-aware Resource Allocation and Scheduling in the Cloud. In *Proceedings of the 3rd USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'11, pages 4–4, Berkeley, CA, USA, 2011. USENIX Association.

[35] Fawaz Paraiso, Nicolas Haderer, Philipe Merle, Romain Rouvoy, and Lionel Seinturier. A Federated Multi-cloud PaaS Infrastructure. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 392–399, June 2012.

[36] Tobias Anstett, Frank Leymann, Ralph Mietzner, and Steeve Strauch. Towards BPEL in the Cloud: Exploiting Different Delivery Models for the Execution of Business Processes. In *Services - I, 2009 World Conference on*, pages 670–677, July 2009.

[37] Xiaoliang Fan, Ruisheng Zhang, and P. Brezillon. Investigating the Feasibility of Making Contexts Explicit in Designing Cloud Workflow. In *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 2121–2128, May 2013.

[38] Tim Dornemann, Ernst Juhnke, and Bernd Freisleben. On-Demand Resource Provisioning for BPEL Workflows Using Amazon's Elastic Compute Cloud. In *Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, CCGRID '09, pages 140–147, Washington, DC, USA, 2009. IEEE Computer Society.

[39] Sebastian Wagner, Oliver Kopp, and Frank Leymann. Towards choreography-based process distribution in the cloud. In *Cloud Computing and Intelligence Systems (CCIS), 2011 IEEE International Conference on*, pages 490–494, Sept 2011.

[40] Tim Dornemann, Ernst Juhnke, Thomas Noll, Dominik Seiler, and Bernd Freisleben. Data Flow Driven Scheduling of BPEL Workflows Using Cloud Resources. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 196–203, July 2010.

[41] Ernst Juhnke, Tim Dornemann, David Bock, and Bernd Freisleben. Multi-objective Scheduling of BPEL Workflows in Geographically Distributed Clouds. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 412–419, July 2011.

[42] Amazon Simple Workflow Service. The Developer Guide. Technical report, January 2012.

[43] Salesforce. Approvals and workflow. http://www.salesforce.com/sales-cloud/workflow-process-software.jsp, 2014.

[44] WSO2 Business Process Server. http://wso2.com/products/business-process-server/, 2014.

[45] Lionel Seinturier, Philippe Merle, Damien Fournier, Nicolas Dolet, Valerio Schiavoni, and Jean-Bernard Stefani. Reconfigurable SCA Applications with the FraSCAti Platform. In *Proceedings of the 2009 IEEE International Conference on Services Computing*, SCC '09, pages 268–275, Washington, DC, USA, 2009. IEEE Computer Society.

[46] Lionel Seinturier, Philippe Merle, Romain Rouvoy, Daniel Romero, Valerio Schiavoni, and Jean-Bernard Stefani. A Component-based Middleware Platform for Reconfigurable Service-oriented Architectures. *Softw. Pract. Exper.*, 42(5):559–583, May 2012.

[47] Cristian Ruz, Françoise Baude, Bastien Sauvan, Adrian Mos, and Alain Boulze. Flexible SOA Lifecycle on the Cloud Using SCA. In *Enterprise Distributed Object Computing Conference Workshops (EDOCW), 2011 15th IEEE International*, pages 275–282, Aug 2011.

[48] Françoise Baude, Denis Caromel, Cédric Dalmasso, Marco Danelutto, Vladimir Getov, Ludovic Henrio, and Christian Pérez. GCM: a grid extension to Fractal for autonomous distributed components. *annals of telecommunications - annales des télécommunications*, 64(1-2):5–24, 2009.

[49] Luciano Resende and Jean-Sebastien Delfino. *Developing Composite Applications for the Cloud with Apache Tuscany.* Apache Foundation, 2013.

[50] The Apache Nuvem incubator. http://incubator.apache.org/nuvem/, 2014.

[51] Tobias Binz, Gerd Breiter, Frank Leyman, and Thomas Spatzier. Portable Cloud Services Using TOSCA. *Internet Computing, IEEE*, 16(3):80–85, May 2012.

[52] Jorge Cardoso, Tobias Binz, Uwe Breitenb ucher, Oliver Kopp, and Frank Leymann. Cloud Computing Automation: Integrating USDL and TOSCA. In *CAiSE 2013*, volume 7908 of *Lecture Notes in Computer Science (LNCS)*, pages 1–16. Springer-Verlag, 2013.

[53] Tobias Binz, Uwe Breitenb ucher, Florian Haupt, Oliver Kopp, Frank Leymann, Alexander Nowak, and Sebastian Wagner. OpenTOSCA – A Runtime for TOSCA - based Cloud Applications. In *11 textsuperscriptth International Conference on Service - Oriented Computing*, LNCS. Springer, 2013.

[54] Thijs Metsch and Andy Edmonds. Open Cloud Computing Interface - RESTful HTTP Rendering. Technical report, 2011.

[55] Nikolaos Loutas, Vassilios Peristeras, Thanassis Bouras, Eleni Kamateri, Dimitros Zeginis, and Konstantinos Tarabanis. Towards a Reference Architecture for Semantically Interoperable Clouds. In *Cloud Computing Technology and Science (CloudCom), 2010 IEEE Second International Conference on*, pages 143–150, Nov 2010.

[56] Benjamin Satzger, Waldemar Hummer, Christian Inzinger, Philip Leitner, and Schahram Dustdar. Winds of Change: From Vendor Lock-In to the Meta Cloud. *Internet Computing, IEEE*, 17(1):69–73, Jan 2013.

[57] Francesco D'Andria, Stefano Bocconi, Jesus Gorronogoitia Cruz, James Ahtes, and Dimitris Zeginis. Cloud4SOA: Multi-cloud Application Management Across PaaS Offerings. In *Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on*, pages 407–414, Sept 2012.

[58] Sami Yangui, Mohamed Mohamed, Samir Tata, and Samir Moalla. Scalable Service Containers. In *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*, pages 348–356, Nov 2011.

[59] Mohamed Mohamed, Sami Yangui, Samir Moalla, and Samir Tata. Web Service Micro-Container for Service-based Applications in Cloud Environments. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2011 20th IEEE International Workshops on*, pages 61–66, June 2011.

[60] Antonio Celesti, Francesco Tusa, Massimo Villari, and Antonio Puliafito. How to Enhance Cloud Architectures to Enable Cross-Federation. In *Cloud Computing (CLOUD), 2010 IEEE 3rd International Conference on*, pages 337–345, July 2010.

[61] Rajkumar Buyya, Rajiv Ranjan, and Rodrigo N. Calheiros. InterCloud: Utility-oriented Federation of Cloud Computing Environments for Scaling of Application Services. In *Proceedings of the 10th International Conference on Algorithms and Architectures for Parallel Processing - Volume Part I*, ICA3PP'10, pages 13–31, Berlin, Heidelberg, 2010. Springer-Verlag.

[62] Sami Yangui and Samir Tata. The SPD approach to deploy service-based applications in the cloud. *Concurrency and Computation: Practice and Experience*, 2014.

[63] Wil van der Aalst. The Application of Petri nets to Workflow Management. *The Journal of Circuits, Systems and Computers*, 8(1):21–26, 1998.

[64] Sebastian Hinz, Karsten Schmidt, and Christian Stahl. Transforming BPEL to Petri Nets. In *Proceedings of the 3rd International Conference on Business Process Management*, BPM'05, pages 220–235, Berlin, Heidelberg, 2005. Springer-Verlag.

[65] Karsten Schmidt. LoLA: A Low Level Analyser. In *Proceedings of the 21st International Conference on Application and Theory of Petri Nets*, International Conference on Agricultural Technology and Plant Nutrition (ICATPN'00), pages 465–474, Berlin, Heidelberg, 2000. Springer-Verlag.

[66] Website of LoLA. http://www2.informatik.hu-berlin.de/top/lola/lola.html, 2014.

[67] Jonathan Billington, Søren Christensen, Kees Van Hee, Ekkart Kindler, Olaf Kummer, Laure Petrucci, Reinier Post, Christian Stehno, and Michael Weber. The Petri Net Markup Language: Concepts, Technology, and Tools. In *Proceedings of the 24th International Conference on Applications and Theory of Petri Nets*, ICATPN'03, pages 483–505, Berlin, Heidelberg, 2003. Springer-Verlag.

[68] Niels Lohmann, Eric Verbeek, Chun Ouyang, and Christian Stahl. Comparing and evaluating Petri net semantics for BPEL. *International Journal of Business Process Integration and Management*, 4(1):60–73, 2009.

[69] Remco M. Dijkman, Marlon Dumas, and Chun Ouyang. Semantics and Analysis of Business Process Models in BPMN. *Inf. Softw. Technol.*, 50(12):1281–1294, November 2008.

[70] Sami Yangui, Kais Klai, and Samir Tata. Deployment of Service-based Processes in the Cloud using Petri Net Decomposition. In *22nd International Conference on COOPERATIVE INFORMATION SYSTEMS (CoopIS 2014)*, October 2014.

[71] B. Bernhard. Web services container, February 13 2003. US Patent App. 10/215,722.

[72] Sami Yangui and Samir Tata. CloudServ: PaaS Resources Provisioning for Service-Based Applications. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 522–529, March 2013.

[73] Aya Omezzine, Sami Yangui, Narjes Bellamine, and Samir Tata. Mobile Service Micro-containers for Cloud Environments. In *Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE), 2012 IEEE 21st International Workshop on*, pages 154–160, June 2012.

[74] Ritu Gupta and Gaurav Kansal. A Survey on Comparative Study of Mobile Agent Platforms. *International Journal of Engineering Science and Technology (IJEST)*, 3(3), 2011.

[75] Mourad Amziani, Kais Klai, Tarek Melliti, and Samir Tata. Time-Based Evaluation of Service-Based Business Process Elasticity in the Cloud. In *Cloud Computing Technology and Science (CloudCom), 2013 IEEE 5th International Conference on*, volume 1, pages 573–580, Dec 2013.

[76] Mohamed Mohamed, Djamel Belaid, and Samir Tata. How to provide monitoring facilities to services when they are deployed in the cloud ? In *CLOSER '12 : 2nd International Conference on Cloud Computing and Services Science*, pages 258–263, Porto, Portugal, 2012. SciTePress. 12533 12533.

[77] Mohamed Mohamed, Djamel Belaid, and Samir Tata. Adding Monitoring and Reconfiguration Facilities for Service-Based Applications in the Cloud. In *Advanced Information Networking and Applications (AINA), 2013 IEEE 27th International Conference on*, pages 756–763, March 2013.

[78] *Service-oriented Computing Track*. IEEE Internet Computing, February 2005.

[79] Running Standalone Web Applications on Cloud Foundry. http://blog.cloudfoundry.org/2012/05/11/running-standalone-web-applications-on-cloud-foundry/, 2014.

[80] Process Types and the Procfile. https://devcenter.heroku.com/articles/procfile, 2014.

[81] Sean Marston, Zhi Li, Subhajyoti Bandyopadhyay, Juheng Zhang, and Anand Ghalsasi. Cloud computing — The business perspective. *Decision Support Systems*, 51(1):176–189, 2011.

[82] Sami Yangui and Samir Tata. An OCCI Compliant Model for PaaS Resources Description and Provisioning. 2014.

[83] Enterprise Service Bus Documentation. Extending the ESB. https://docs.wso2.org/display/ESB460/Extending+the+ESB, 2014.

[84] Heroku PaaS Add-ons. https://addons.heroku.com/, 2014.

[85] Cloud Foundry Marketplace. https://run.pivotal.io/#marketplace, 2014.

[86] A generic Cloud Application Provisioning and Management API. http://www-inf.it-sudparis.eu/SIMBAD/tools/COAPS/, 2014.

[87] The SPD approach to deploy service-based applications. http://www-inf.int-evry.fr/SIMBAD/tools/SPD/, 2014.

[88] Sami Yangui, Marwa Ben Nasrallah, and Samir Tata. PaaS-Independent Approach to Provision Appropriate Cloud Resources for SCA-based Applications Deployment. In *Semantics, Knowledge and Grids (SKG), 2013 Ninth International Conference on*, pages 14–21, Oct 2013.

[89] Mohamed Sellami, Sami Yangui, Mohamed Mohamed, and Samir Tata. PaaS-Independent Provisioning and Management of Applications in the Cloud. In *IEEE CLOUD*, pages 693–700, 2013.

[90] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis.

[91] Jean-Pierre Laisne, Iaines James Marshall, and Patrick Peiravi. Next-Generation Cloud Management. The CompatibleOne Project. *Intel's Journey to Cloud*, 2.0(1):15–24, 2012.

[92] Sami Yangui, Iain-James Marshall, Jean-Pierre Laisne, and Samir Tata. CompatibleOne: The Open Source Cloud Broker. *Journal of Grid Computing*, 12(1):93–109, 2014.

[93] Easi-Clouds project description. http://easi-clouds.eu/2012/02/03/project-description/, 2014.

[94] Janne Lautamäki, Antti Nieminen, Johannes Koskinen, Timo Aho, Tommi Mikkonen, and Marc Englund. CoRED: Browser-based Collaborative Real-time Editor for Java Web Applications. In *Proceedings of the ACM 2012 Conference on Computer Supported Cooperative Work*, CSCW '12, pages 1307–1316, New York, NY, USA, 2012. ACM.

[95] Eman Hossny, Sherif Khattab, Fatma Omara, and Hesham Hassan. A Case Study for Deploying Applications on Heterogeneous PaaS Platforms. In *Cloud Computing and Big Data (CloudCom-Asia), 2013 International Conference on*, pages 246–253, Dec 2013.

[96] Srinath Perera, Chathura Herath, Jaliya Ekanayake, Eran Chinthaka, Ajith Ranabahu, Deepal Jayasinghe, Sanjiva Weerawarana, and Glen Daniels. Axis2, Middleware for Next Generation Web Services. In *Proceedings of the IEEE International Conference on Web Services*, ICWS '06, pages 833–840, Washington, DC, USA, 2006. IEEE Computer Society.

[97] Etienne Langlet. *Apache Tomcat 6 Guide d'administration du serveur Java EE sous Win- dows et Linux.* . Eni edition edition, 2008.

[98] Arulazi Dhesiaseelan and Venkatavaradan Ragunathan. Web Services Container Reference Architecture (WSCRA). In *Proceedings of the IEEE International Conference on Web Services*, ICWS '04, pages 806–, Washington, DC, USA, 2004. IEEE Computer Society.

[99] Robert L. Grossman. The Case for Cloud Computing. *IT Professional*, 11(2):23–27, March 2009.

[100] James Caverlee, Calton Pu, Dimitrios Georgakopoulos, and James Joshi. Editorial for CollaborateCom 2011 Special Issue. *MONET*, 18(2):235–236, 2013.