

soCloud: A service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds

Fawaz Paraiso, Philippe Merle, Lionel Seinturier

▶ **To cite this version:**

Fawaz Paraiso, Philippe Merle, Lionel Seinturier. soCloud: A service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds. Computing, 2016, 98 (5), pp.539-565. 10.1007/s00607-014-0421-x . hal-01019385

HAL Id: hal-01019385
https://inria.hal.science/hal-01019385

Submitted on 7 Jul 2014

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Noname manuscript No.
(will be inserted by the editor)

soCloud: A service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds

Fawaz Paraiso · Philippe Merle · Lionel Seinturier

Received: 12 th July, 2013 / Accepted: date

Abstract Multi-cloud computing is a promising paradigm to support very large scale world wide distributed applications. Multi-cloud computing is the usage of multiple, independent cloud environments, which assumed no priori agreement between cloud providers or third party. However, multi-cloud computing has to face several key challenges such as *portability*, *provisioning*, *elasticity*, and *high availability*. Developers will not only have to deploy applications to a specific cloud, but will also have to consider application portability from one cloud to another, and to deploy distributed applications spanning multiple clouds. This article presents soCloud a service-oriented component-based Platform as a Service (PaaS) for managing portability, elasticity, provisioning, and high availability across multiple clouds. soCloud is based on the OASIS Service Component Architecture (SCA) standard in order to address portability. soCloud provides services for managing provisioning, elasticity, and high availability across multiple clouds. soCloud has been deployed and evaluated on top of ten existing cloud providers: Windows Azure, DELL KACE, Amazon EC2, CloudBees, OpenShift, dotCloud, Jelastic, Heroku, Appfog, and an Eucalyptus private cloud.

Keywords Multi-cloud computing · Platform as a Service · Portability · Provisioning · Elasticity · High availability · Service Component Architecture

Fawaz Paraiso
Inria Lille - Nord Europe & University Lille 1
LIFL UMR CNRS 8022, France
E-mail: fawaz.paraiso@inria.fr

Philippe Merle
Inria Lille - Nord Europe & University Lille 1
LIFL UMR CNRS 8022, France
E-mail: philippe.merle@inria.fr

Lionel Seinturier
Inria Lille - Nord Europe & University Lille 1
LIFL UMR CNRS 8022, France
IUF - Institut Universitaire de France
E-mail: lionel.seinturier@inria.fr

1 Introduction

Cloud computing builds on established trends for driving the cost out of the delivery of services while increasing the speed and agility with which services are deployed. Virtualization, on-demand deployment, Internet delivery of services are parts of Cloud computing. Cloud computing differentiates itself by changing how we invent, develop, deploy, scale, update, maintain, and pay for applications and the infrastructure on which they run.

Different cloud service providers, based on different technologies, support a large number of cloud services such as Infrastructure as a Service (IaaS) and Platform as a Service (PaaS). Cloud service consumers select what fit their requirements from the cloud services. For instance, requirements can be: price, quality of service (QoS), programming language, database, middleware, etc. It is difficult to cloud service consumers to meet all these requirements with a single cloud provider. Multi-cloud computing as the usage of multiple, independent cloud environments, which assumed no priori agreement between cloud providers or third party is a promising paradigm to support very large scale world wide distributed applications.

However, multi-cloud computing has to face several key challenges: *portability*, *provisioning*, *elasticity*, and *high availability*. Multi-cloud portability means writing applications once and running them on any clouds. Most existing cloud providers are typically offered through proprietary APIs and limited to a single infrastructure provider. In such situations, vendor lock-in is a primary concern for moving towards a cloud provider. Multi-cloud provisioning refers to the capability to deploy a distributed application spanning multiple cloud providers. Deploying a distributed application in a multi-cloud context is not an easy task. Multi-cloud elasticity refers to the capability to scale applications across multiple clouds. Currently, there is no convenient way to express specific application elasticity rules for each part of a distributed application as needed. Multi-cloud high availability refers to the degree to which an application is operable across multiple clouds. Cloud provider services can become unavailable due to outages or denials of services. High availability needs to be analysed and set across multiple clouds in order to reduce the probability of outages that could affect services deployed in a single cloud system.

In this article we discuss the design and implementation of soCloud. soCloud is a multi-cloud PaaS that addresses the four key challenges presented previously. soCloud is a distributed PaaS that provides a model for building distributed applications. This model is an extension of the OASIS SCA standard[1]. Our ongoing approach to address portability and provisioning in a multi-cloud context is the use of the SCA standard. Our elasticity management approach is based on autonomic computing with the overall aim of creating self-managed elastic multi-cloud applications. High availability is achieved in two ways. Firstly, soCloud provides a multi-cloud load balancer service that fronts traffic for applications deployed across multiple clouds and makes a decision about where to route the traffic when cloud nodes fail. Secondly, the soCloud architecture uses redundancy at all levels to ensure that no single component failure in a cloud provider impacts the overall system availability. We describe a way

[1] http://www.oasis-opencsa.org/sca

to annotate SCA artifacts with deployment information needed to optimize the use of services in multiple cloud environments. These annotations also allow to express elasticity rules that ensure the appropriate adjustment decisions made in timely manner to meet service needs in the presence of cloud service failures. The soCloud architecture is composed of the following SCA components: service deployer, constraints validator, PaaS deployment, SaaS deployment, load balancer, node provisioning, monitoring, workload manager and controller components. soCloud is deployed and evaluated on ten existing cloud providers Windows Azure, DELL KACE, Amazon EC2, CloudBees, OpenShift, dotCloud, Jelastic, Heroku, Appfog, and an Eucalyptus private cloud.

The remainder of this article is organized as follows. In Section 2, we discuss the four challenges we addressed for multi-clouds. Next, Section 3 presents the design and implementation of the soCloud platform, and its integration with existing cloud providers. The evaluation of soCloud is discussed in Section 4. Section 5 compares soCloud with the state-of-the-art. Section 6 discusses the limitations of this work, while Section 7 concludes this article and presents future work we intend to address.

2 Multi-cloud challenges

IT companies are starting to realize and recognize the benefits and advantages of cloud computing. However, cloud technology maturity is still a concern. This section describes four key challenges for multi-cloud computing: *portability*, *provisioning*, *elasticity*, and *high availability*.

2.1 Multi-cloud portability

In the cloud computing area, the portability issue should take into account both *application* and *data*. Although data portability is an important feature, this article focuses only on application portability. In an emerging and rapidly changing market such as cloud computing, it is easy to create applications that are locked into one vendor cloud because of the use of proprietary APIs and formats. To avoid this vendor lock-in syndrome, SaaS must be portable on top of various cloud PaaS and IaaS providers. Then, this multi-cloud portability allows the migration from one cloud provider to another in order to take advantage of cheaper prices or better QoS. However, SaaS portability requires that the runtime support provides a common model to hide the diversity of underlying PaaS and IaaS. Furthermore, the dominant programming models today have grown increasingly complex. SCA, in contrast, provides a simplified programming model and unified way to applications that communicate using a variety of network protocols [28].

To address the challenge of *multi-cloud portability*, soCloud promotes SCA as the model to design and develop both multi-cloud SaaS applications and the underlying soCloud PaaS.

2.2 Multi-cloud provisioning

Application Provisioning: Application provisioning includes building and deployment on multiple cloud environments. Providing a consistent methodology and process for modelling how applications are built and provisioned, enabling flexibility and

choice for developers to use any cloud provider they choose. Application provisioning should deliver business agility and operational efficient by high level abstraction and automating provisioning of applications across multiple cloud providers.

Geo-diversity: The authors in [43] advocates that small data centers, which consume less power, may be more advantageous than large ones, and that geo-diversity tends to better match user demands. Geo-diversity lowers latency to users and increases reliability in the presence of an outage taken out an entire site. In a legal context, data protection law and confidentiality can lead users to place their data in a specific area. In fact, the location of data can be facilitated or restricted in particular jurisdictions.

Overall, to address the challenge of *multi-cloud provisioning*, soCloud offers a service to provision applications across multiple cloud providers.

2.3 Multi-cloud elasticity

The management of elasticity can be further split into two approaches: fine-grained or coarse-grained. The first one allows to scale resources either by changing the number of virtual machines (VMs) using horizontal scaling (adding more virtual machines or devices to the computing platform to handle an increased application load) or vertical scaling (adding more CPU, Memory, Disk, Bandwidth to handle an increased application load) depending on the application memory, storage, network bandwidth and CPU requirements. The second one manages the resources scalability by changing cloud providers. Indeed, when outages occur with one cloud provider, the coarse-grained elasticity will switch to another cloud provider. While, the fine-grained elasticity can actually be made up of many fine-grained resources. In managing elasticity across multiple clouds, automation is a mandatory requirement, and it is thus a foundational design principle [23]. The function of any autonomic capability is a control loop that collects details from the system and acts accordingly. However, developers should have the possibility to define specific elasticity rules on their services. For example, the developers specify constraints on the response time depending of the number of users currently accessing the provided service.

To address the challenge of *multi-cloud elasticity*, soCloud offers an autonomic service which provides a global mechanism to manage elasticity across multiple clouds and also offers the possibility to define application specific elasticity rules.

2.4 Multi-cloud high availability

A series of news [22,42] and papers [3,36] have pointed several cloud provider outages. According to a recent report by the International Working Group on Cloud computing Resiliency[2], a total of 568 hours of downtime at thirteen well-known cloud services since 2007 caused financial damage of more than US$71.7 million. The average unavailability of cloud services is 7.5 hours per year, amounting to an availability rate of 99.9%, according to the group preliminary results. These results are far from the expected reliability of mission critical system which is 99.999%. As a comparison, the average unavailability for electricity in a modern capital city is less than 15 minutes per year [30]. Besides this economic impact, the downtime

[2] http://iwgcr.org

also affects millions of end-users. Of course, downtime costs money and dammage, unfortunately protecting systems against downtime with 99.999% of availability is not free.

To address the challenge of *multi-cloud high availability* despite outages, soCloud provides high availability in two ways. Firstly, with the applications deployed with a soCloud platform, the high availability is ensured by using a load balancer service which distributes requests among instances of the application deployed on multiple cloud providers. Secondly, the soCloud architecture uses redundancy at all levels to ensure that no single component failure in a cloud provider impacts the overall system availability.

3 soCloud design and implementation

In this section we present the design and implementation of soCloud platform. We first discuss background elements of SCA and FraSCAti. Next, we describe some components of the soCloud architecture and its implementation. Finally, we describe how the soCloud platform is deployed on existing IaaS/PaaS providers.

3.1 SCA

soCloud is based on the SCA standard. SCA is a set of OASIS specifications for building distributed applications and systems using Service-Oriented Architecture (SOA) principles [15]. SCA promotes a vision of Service-Oriented Computing (SOC) where services are independent of implementation languages (Java, Spring, BPEL, C++, COBOL, C, etc.), networked service access technologies (Web Services, JMS, etc.), interface definition languages (WSDL, Java, etc.) and non-functional properties. Component-Based Design [11] and SOA are two major software engineering approaches widely used for structuring systems. SCA targets composition of services in SOA systems and thus is suitable for building enterprise and cross-enterprise applications built on already-developed components and services.

3.2 FraSCAti

Several open source implementations of the SCA specifications exist. Three of the most well known are Apache Tuscany, Fabric3 and FraSCAti. Compared to Tuscany and Fabric3, FraSCAti introduces reflective capabilities to the SCA programming model, and allows dynamic introspection and reconfiguration via a specialization of the Fractal component model [6]. FraSCAti provides a component-based approach to support the heterogeneous composition of various interface definition languages (WSDL, Java), implementation technologies (Spring, EJB, BPEL, OSGI, Jython, Jruby, Xquery, Groovy, Velocity, Fscript, Beanshell.), and binding technologies (Web Services, JMS, RPC, REST, RMI, UPnP.).

soCloud is built on top of FraSCAti. FraSCAti is the execution environment of both the soCloud PaaS and soCloud applications deployed on the top of this multi-cloud PaaS.

3.3 soCloud SaaS applications

Application specification soCloud applications are built using the SCA model. As illustrated in Fig. 1, the basic SCA building blocks are software components, which

provide services, require references and expose properties. The references and services are connected by wires. For SCA references, a binding describes the access mechanism used to invoke a remote service. In the case of services, a binding describes the access mechanism that clients use to invoke the service. We describe how SCA can be used to package SaaS applications. The first requirement is that the package must describe and contain all artifacts needed for the application. The second requirement is that provisioning constraints and elasticity rules must be described in the package. The SCA assembly model specification describes how SCA and non-SCA artifacts (such as code files) are packaged. The central unit of deployment in SCA is a contribution. A contribution is a package that contains implementations, interfaces and other artifacts necessary to run components. The SCA packaging format is based on ZIP files, however, other packaging formats are explicitly allowed. Fig. 1 shows a three-tier application is packaged as a ZIP file (SCA contribution) and its architecture is described.

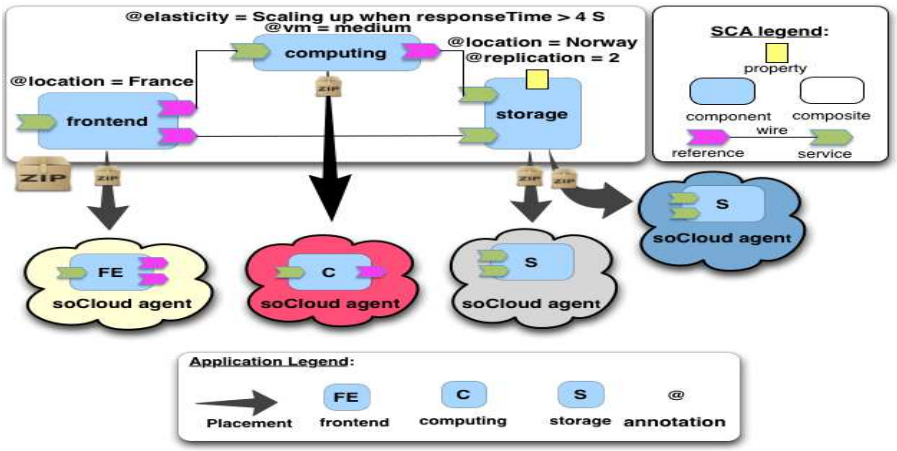


Fig. 1 An annotated soCloud application.

Annotations Some cloud-based applications require more detailed description of their deployment (c.f. Fig. 1). The deployment and monitoring of soCloud applications are bound by a description of the overall software system architecture and the requirements of the underlying components, that we refer to as the *application manifest*. Basically, the *application manifest* consists of describing what components the application is composed with functional and non-functional requirements for deployment. In fact, the application can be composed of multiple components (c.f. Fig. 1). The application manifest defines elasticity rule for the service component (e.g., increase/decrease instance of component). Commonly, scale up or down, is translated to a **condition-action** statement that reasons on performance indicators of the component deployed. In order to fulfill the requirements for the soCloud application descriptor, we propose to annotate the SCA components with the four following annotations:

1. **placement constraint** (@*location*) allows to map components of a soCloud application to available physical hosts within a geographical datacenter in multi-cloud environments.
2. **computing constraint** (@*vm*) provides necessary computing resources defined for components of a soCloud application in the multi-cloud environments.
3. **replication** (@*replication*) specifies the number of instances of the component that must be deployed in multi-cloud environments.
4. **elasticity rule** (@*elasticity*) defines a specific elasticity rule that should be applied to the component deployed on multi-cloud environments.

For example, let us consider the three-tier web application described in Fig. 1. The annotation (@*location=France*) of the frontend component indicates to deploy this component on a cloud provider located in France. Next, the annotation (@*vm=medium*) on the computing component specifies the kind of computing resources required by this component and can be deployed on any cloud provider. The developer has the possibility to specify through the @*vm* annotation the computing resources (micro, small, medium, large) she need. Finally, the annotations (@*location=Norway* and @*replication=2*) on the storage component indicate to deploy this component on two different cloud providers located in Norway. soCloud automates the deployment of this three-tier application in a multiple cloud environment by respecting given annotations.

3.4 Constraint analysis and formulation

To express constraints (placement, computation, etc.) and define specific elasticity rules, we analyse each step of the formulation of these constraints. To express a constraint we use this formula P = {n, v}. Where *n* indicates the name of the constraint and *v* the value of the constraint. Regarding the elasticity rule, we use R = {c, a}, where *c* indicates the condition and *a* the resulting action.

The placement can be a location or a provider name. For example, specifying @*placement="Amazon_Ireland"* or @placement="Ireland" on a component has the same interpretation (i.e, the component should be placed in Ireland) from the point of view of *placement constraint*.

To express different computation capacities, we use the instance type taxonomy defined by Amazon EC2[3]. An example of computation constraint request is P = {vm, medium}. This request has *name* "vm" and value "medium" which represents the computing capacity.

However, in a heterogeneous multi-cloud environment, existing types of VM provided by different clouds can have small differences. In order to use these clouds and hide these differences, the soCloud platform defines a high-level abstraction where similar VMs are classified in the same type. Let T be the set of VM types defined in the soCloud platform (see Equation 1) and C the characteristics of the VM provided by different clouds (see Equation 2). Equation 3 defines how the soCloud platform hides the differences between the VMs provided from different clouds.

$$T = \{micro, small, medium, large\} \tag{1}$$

[3] http://aws.amazon.com/ec2/instance-types/

$$C = \{C_i, i \in Provider\} \tag{2}$$

$$VM_{type} = \{C : f(C_i), f(C_i) \in T, i \in Provider\} \tag{3}$$

The soCloud platform offers to developers the opportunity to choose the specific VM (with the best performance) by indicating, as an additional information, *the provider name*. Overall, choosing a specific VM refers to the combination of *@vm* and @placement annotations, where @placement corresponds to the provider name.

3.5 A soCloud application descriptor

Let us consider the three-tier application architecture described in Fig. 1, where we need to deploy a distributed application. This distributed application is packaged as a contribution that contains three contributions and one file (*application descriptor*) describing the architecture of the distributed application. Each contained contribution corresponds to each tier of the distributed application. In the case the application deployed is not distributed, the contribution contains a single contribution with a composite file. This distributed application needs placement, elasticity, and computation requirements. In order to fulfill these requirements, we use SCA properties to express them (see Listing 1). Lines 3, 9, 18 correspond to our SCA extension defined to represent respectively frontend, computing and storage contributions. The placement constraints for frontend and storage components are expressed at Line 6 and 20 respectively. The computing constraint is expressed at Line 12 for the computing component. The number of replication of the storage component is expressed at Lines 21. Lines 13-15 express the elasticity rule for the computing component. We adopt an event-condition-action approach for rule specification. The event-condition syntax is an Event Processing Language statement [25]. Basically, the elasticity rule and action defined at Lines 13-15 means: when the average response time of the component exceeds 4 seconds, then add a new virtual machine running this component.

```
1  <composite name="DistributedApplication">
2    <component name="frontend">
3      <implementation.contribution contribution="frontend.zip"/>
4      <reference name="compute" target="computing/compute"/>
5      <reference name="storage" target="storage/storage"/>
6      <property name="location">France</property>
7    </component>
8    <component name="computing">
9      <implementation.contribution contribution="computing.zip"/>
10     <service name="compute"/>
11     <reference name="storage" target="storage/storage"/>
12     <property name="vm">medium</property>
13     <property name="elasticity">
14     Scaling out when ResponseTime > 4s
15     </property>
16   </component>
17   <component name="storage">
18     <implementation.contribution contribution="storage.zip"/>
19     <service name="storage"/>
20     <property name="location">Norway</property>
21     <property name="replication">2</property>
22   </component>
23 </composite>
```

Listing 1 A soCloud application descriptor.

3.6 soCloud architecture

Fig. 2 gives an overview of the soCloud architecture. The soCloud architecture has two parts: the soCloud *master*, and the soCloud *agent*. This partitioning provides flexibility for deploying the soCloud PaaS across a highly distributed multi-cloud environment. Firstly, the soCloud *master* consists of a set of eight components. This part of the architecture focuses on the intelligence processing of soCloud. Secondly, the soCloud *agent* is used to host, execute and monitor soCloud applications. This part provides the necessary services for managing a set of applications and resources. soCloud *agents* work with the soCloud *master* and run in different cloud infrastructures. All communication between a soCloud *master* and the applications deployed is mediated by the soCloud *agent*.

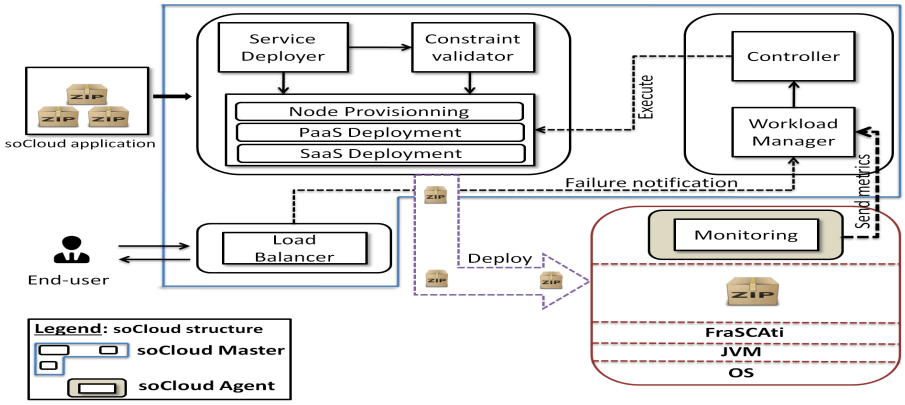


Fig. 2 Overview of the soCloud Architecture.

3.6.1 Monitoring

The component provides an unified-platform independent mechanism that collects, aggregates and reports details (such as health and performance metrics) about applications deployed on multiple cloud environments. It brings information about currently executing process as well as the system on which the monitoring service is running. The *monitoring* component captures any change in the state of the application. The monitoring associates to each application deployed on the soCloud platform, a temporary table (ResponsivenessEvent) that collects informations such as application *responseTime*, *number of requests*, etc. The metrics collected in a time interval are sent to the *Workload Manager* component for analyzing. The monitoring component acts at three levels: Operating System (OS), Java Virtual Machine (JVM), and Execution Environment (FraSCAti). The monitoring component exposes services via REST and JMS to monitor a distributed environment. The consumer of these services is the *workload manager* component or can be also any external application running outside soCloud. Each application deployed with the soCloud PaaS is automatically monitored. However, with some cloud providers such as Salesforce.com or Google

App Engine our monitoring component could not work. As example, Google App Engine forbids the use of JMX.

3.6.2 Workload Manager

The Workload Manager (WM) component provides some event processing functionality [16]. All events are processed to extract drift indicators (DI). An example of DI can be a CPU consumption is greater than 90% for a period of 2 minutes. The WM is centered on DI tracking perform filtering, transformation, and most importantly aggregation of events. All the metrics (events data) sent by *monitoring* components are continuously analyzed in terms of drift indicators that are expressed by event rules, and acts upon opportunities and threats in real time, potentially by creating derived events. One of WM major goals is to find a symptom and analyses it to find its root cause. The WM uses a technique called event *correlation*[4] to examine symptoms and identify groups of symptoms that have a common root cause. As an example of event correlation, WM takes multiple occurrences of the same event, examines them for duplicate information, removes redundancies and reports them as a single event. When a drift occurs, the WM reports it to the *controller* component. The ability to derive instant insights into the operations of the resource provisioning is essential. Thus, the capability to dynamically allocate and dispose resources is an important ingredient to build a platform for elastic applications.

Related to the events received from an inbound *monitoring* component, how events can be woven together to pull out the right information? This is accomplished through Complex Event Processing (CEP)[5]. To achieve this, we use DiCEPE, a Distributed Complex Event Processing Engine we have presented in [35]. The particularity of DiCEPE is the integration of CEP engines in distributed systems, and the fact that they can be exposed via various communication protocols. The DiCEPE integrates the Esper[6] engine for further processing. We apply Esper because of its performance and the metric-value pairs are delivered as events each time their values change between measurements.

3.6.3 Controller

The *controller* component provides the mechanisms that construct the actions needed to achieve goals and objectives. For example, it multiplexes workloads onto an existing infrastructure, and allows for on-demand allocation of resources to workloads. The system state is managed by the *controller* component. By state, we mean the information retained in one component that is meaningful for this component (as example: a table on each instance of LB to associate network addresses with the symbolic names of available hosts). The system state offers the potential for improving the consistency, and reliability of the system. For components to work together effectively, they must agree on common goals and coordinate their actions. This requires that each part to know something about the other. For example, the *node provisioning* stores a table of available resources: If the developer wants to deploy an application

[4] http://tinyurl.com/qdrcpm3.

[5] **CEP**: Computing that performs operations on complex events, including reading, creating, transforming, or abstracting them[25].

[6] http://www.espertech.com/

on the resource, the controller can notify the *node provisioning* to allocate new resources for the application when the available resource is not sufficient. The second potential advantage of the system state is reliability. If information is replicated at several cloud providers and one of the copies is lost due to a failure, then it may be possible to use one of the other copies to recover the lost information. Compared to the *workload manager* and *node provisioning* components, the *controller* takes decision in the system. The *controller* component is self-adaptive in order to respond in a coherent and timely manner to changes in environment, and to failures of components.

All requests handled by a *controller* component are processed as transactions. The transaction engine is implemented for the specific needs of the soCloud architecture. Each transaction is created and managed by a coordinator. Two well-known problems of concurrent transactions can be mentioned: i) lost update and ii) inconsistent retrievals. To avoid these problems we use a serially equivalent execution of transactions [12]. The use of serial equivalence as a criterion for correct concurrent execution prevents the occurrence of lost updates and inconsistent retrievals. The *controller* component is the core of the elasticity management, it is made to tolerate failures by the use of redundant components.

3.6.4 Service Deployer

The process illustrated by the sequence diagram in Fig. 3 describes how each task vary with a service deployment scenario. The *Service Deployer* (SD) component is responsible for handling the additional information of coordinating and managing the service across multiple clouds (i.e., placement, binding, manage service). The SD component decomposes and captures the constraints (specified by the developer) of the service. For example, a constraint can be a placement of an application, resource capacities needed by an application, or defined elasticity rules. In the case where the constraints expressed on the components are fulfilled by multiple cloud providers, soCloud randomly choses one of the providers offering the lowest price. To perform

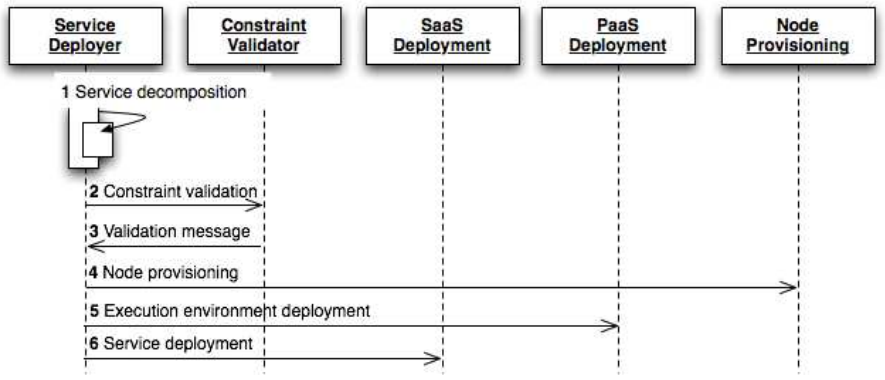


Fig. 3 Application deployment sequence diagram.

the deployment, the SD component captures the constraints defined and validates

them with the *Constraint Validator* component. Once the validation is done, the SD component deploys a whole application this corresponds to sequences 4 to 6 in Fig. 3 with the support of the SaaS Deployment, PaaS Deployment, and Node Provisioning components.

This component deploys the contribution package in three steps:

1. Validates the contribution package by checking if the contribution package contains at least one ZIP file and one composite file.
2. Uses the constraint validator to validate the SCA properties defined in the composite file.
3. Matches each constraint or elasticity rule defined in the composite file and invokes the corresponding execution operation: *Node provisioning*, *PaaS deployment*, *SaaS deployment*.

3.7 Elasticity specification

In this section, we will describe how the soCloud architecture automates specific elasticity rules associated with soCloud applications.

soCloud manages elasticity at IaaS and PaaS levels in the same manner. In fact, the elasticity management in soCloud is not focused on any cloud layer (IaaS or PaaS) specific resources, instead it refers to resources through abstractions provided by the NP component, that offers an uniform way to manage resources from both IaaS and PaaS. soCloud provides the capacity to scale the resources allocated for the application as needed. For example, soCloud can add more nodes if it detects a degradation on the application performance. On the other hand, if the resources are underused, resizing is necessary. This feature is managed as a feedback control loop by the soCloud platform. However, for specific cases, the developer should be able to define automatic elasticity rules associated to its application. These rules are defined inside the application architecture and supervised by the soCloud platform. Each rule is composed of a condition or a set of conditions to be monitored. Those specific elasticity rules are also managed by the soCloud feedback control loop. In order to achieve elasticity, we need to keep track of the frequency of requests to resources hosted and applications deployed on them. Thus, we use a *proactive scheme* that relies on the current workload arrival rate to detect overload conditions. We measure the incoming workload rate by monitoring the number of user connections being opened in the load balancer component. To maintain hit statistics for frequently-accessed applications, we dynamically compute an exponential weighted moving average of request inter-arrival times, along the same lines as TCP computes its estimated round-trip time [24]. Specially, we compute an average of the inter-arrival time using the following formula:

$$f(t) = (1 - \alpha) * f(t-1) + \alpha * (\delta t(t) - \delta t\,(t-1)) \tag{4}$$

The arrival time of every hit is represented by $\delta t(t)$. The constant α is a smoothing factor that puts more weight on recent samples than on old samples. We have used a value of α = 0.125, which is recommended for TCP[7].

[7] http://tools.ietf.org/html/rfc2988

To detect overloads and underloads in the soCloud platform, we use a *threshold-based* scheme to trigger dynamic allocation. Let us note that the calculation of a *threshold* scheme based on equation 4 varies from one deployed application to another.

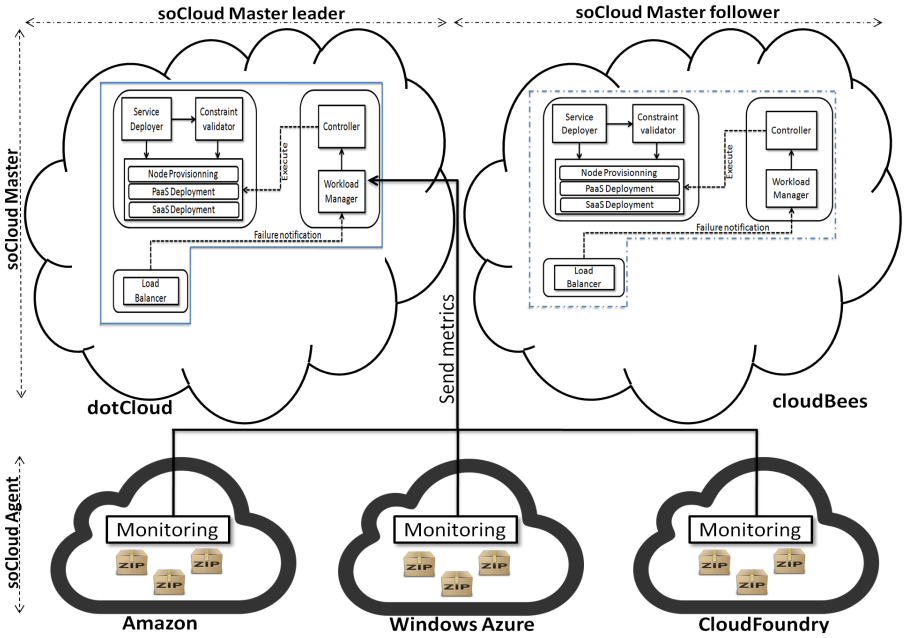


Fig. 4 Conceptual view of a soCloud deployment.

3.8 soCloud deployment

In this section we describe how the soCloud architecture is deployed on a concrete multi-cloud environment.

The soCloud PaaS provides high availability by replicating itself on different clouds as shown in Fig. 4. We assume in our implementation that cloud services can fail, and such fault service may later recover. The system administrator has the possibility to define the deployment policy by specifying the number of replications. For instance in Fig. 4, there is one replication of the soCloud master. Then, the deployment is done in three steps. In the first step, the soCloud *master* is deployed in dotCloud. In the second step, the soCloud *master* (deployed in dotCloud) dynamically deploys another soCloud *master* in CloudBees. Automatically, the first soCloud *master* becomes leader and the second one the follower. The soCloud *master* leader is active, while the soCloud *master* follower is passive. By active, we mean the soCloud *master* processes the operations in the system. By passive, we refer to the standby soCloud *master* used as replication. At this stage, only the soCloud *master* and its replication are deployed. Finally, the soCloud *master* leader will provision a new cloud node on which it deploys both the execution environment (FraSCAti) and a soCloud *agent*. When the soCloud *agent* is deployed, it uses a service discovery mechanism to find which *Workload Manager* component the information collected

should be sent. Periodically, the service discovery checks if the *Workload Manager* component is reachable in order to update the services table when failure occurs on the target soCloud *master*. The soCloud PaaS service discovery mechanism is implemented using Google Fusion Table [19] and FraSCAti dynamical multiple reference binding. We use *Google Fusion Table* to persist the state of the active master and with the *dynamical multiple reference* provided by FraSCAti we add on the fly a reference to the new component. By state we mean the operational state of the soCloud *master*. soCloud provides a capability for reliability using sources of state that are external to soCloud itself. Typically, this is done with *Google Fusion Table*. The soCloud platform provides a mechanism called "health checking" by which a component notifies its health. This mechanism is implemented as an XML push mechanism which tests if a component is reachable. Both the soCloud *master* and *agent* need the execution environment (FraSCAti) to be running. However, when the system grows (the number of applications or load increase), the third step is repeated.

3.9 Fail-overs

In this section we describe how the soCloud PaaS ensures the high availability at two levels: **soCloud level**, and **application level**.

soCloud level The active soCloud *master* is called the leader and the passive soCloud *master* is called the follower. The process of electing a leader allows the system to indicate which soCloud *master* will have the decision of execution. The soCloud *master* leader and follower are synchronized such that when the leader fails, automatically the leader election is organized to elect a new leader. Specifically, we use *Wait-Free Synchronization* that is appropriate in fault tolerant and real-time applications [18]. In the case the system administrator has been defined only one replication of the soCloud *master*, the soCloud *master* follower is automatically elected. Otherwise, the election is organized between the soCloud *master* followers. The leader election is organized and supervised by the *controller* component. We assume that each component has a *reachable latency*. The *reachable latency* is obtained by making a ping from one component to another. Ping refers to the ability to have a live component connection. Our leader election algorithm is simple. This algorithm ensures that the component with minimum reachable latency gets elected as the leader. However, the soCloud platform is not restricted to this algorithm, the system administrator has the possibility to define another one (e.g., Chang-Roberts algorithm [9], Malpini algorithm [26]), according to her requirement. Elections are held between two entities that have the same function (e.g., two monitoring components, two workload manager components, etc.). Then, the *controller* component organizes an election in order to compare the *reachable latency*. By using this strategy, all the components of the soCloud *master* leader are in the same cloud and the follower components in other. When the soCloud *master* follower fails, automatically the soCloud master leader deploys a new soCloud *master* follower.

Application level Same to soCloud replication, the developer has the possibility to define the number of instances which will be deployed for its application. Each appli-

cation deployed with soCloud is replicated in different clouds. The fail-overs mechanism is achieved by the *LB* component. When failure occurs with one instance of the application, the *Controller* component takes the decision to instantiate a new one.

Overall, the fail-overs automation in the soCloud platform enables our system to recover quickly from most outages. In addition, we also monitor our system for any variety of error conditions. With the two levels of availability, the soCloud PaaS addresses the high availability challenge presented in Section 2.4.

3.10 Recovery

In this section we describe the method used by the soCloud PaaS for fault tolerance, i.e., *check-pointing*[8]. A checkpoint can be local to a process or global in the system. With the soCloud PaaS we use a global checkpoint. We use Google Fusion Table [19] to record a global state of the system so that in the event of failure the entire system can be rolled back to the global checkpoint and restarted. To record the global state, soCloud uses the coordinated checkpoint method [5]. In fact, there are some disadvantages of uncoordinated checkpoint compared with coordinated checkpointing schemes [18]. First, for coordinated checkpoint it is sufficient to keep just the most recent global state snapshot in the stable storage. For uncoordinated checkpoints a more complex processing scheme is required. Moreover, in the case of a failure, the recovery method for coordinated checkpoint is simpler.

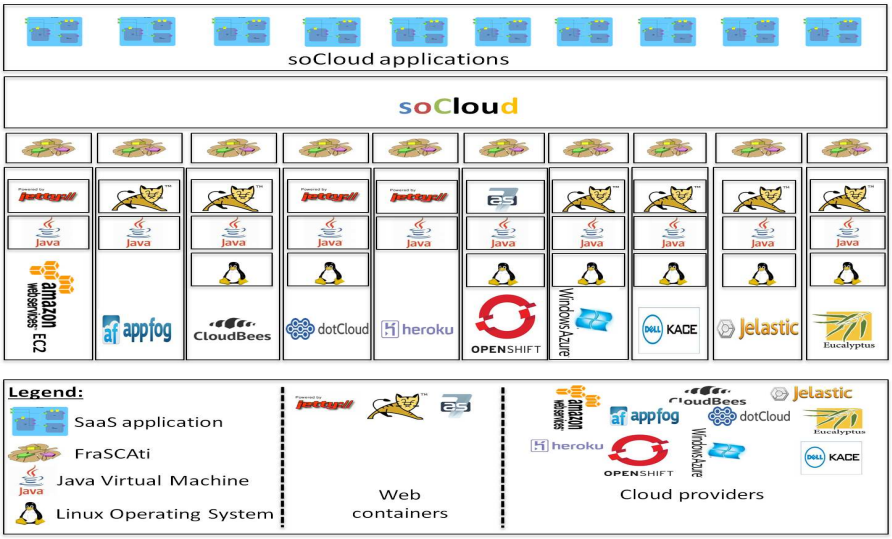


Fig. 5 soCloud deployment with ten cloud providers.

[8] A checkpoint is a snapshot of the state of a process, saved on nonvolatile storage to survive process failures [41].

3.11 Integration with existing IaaS/PaaS

We report on the existing cloud environments on which the soCloud platform has been deployed. The soCloud platform extends an experiment that was presented in a previous work [34]. The soCloud platform is actually deployed on ten target cloud environments that are publicly accessible on the Internet[9]. The deployment is done with IaaS/PaaS providers as illustrated in Fig. 5. With IaaS, resources are provisioned from Windows Azure, DELL KACE, Amazon EC2, and our Eucalyptus private cloud, we installed a PaaS stack composed of a Linux distribution, a Java Virtual Machine, a web container and FraSCAti. soCloud is also deployed on PaaS such as: CloudBees, OpenShift, dotCloud, Jelastic, Heroku, and Appfog as a WAR file.

4 Evaluation

In this section, we evaluate three key aspects of the soCloud platform: *elasticity*, *high availability* and the overhead introduced by soCloud. Firstly, Section 4.1 describes a use case scenario. Then, Section 4.2 evaluates the reaction of soCloud when faced with flash crowd effects (i.e., elasticity of soCloud). Section 4.3 evaluates the soCloud behavior against failures (i.e., high availability of soCloud). Finally, Section 4.4 evaluates the overhead introduced by soCloud.

4.1 Use case

We describe a scenario that can be used in a multiple clouds environment, and explain briefly its requirements.

4.1.1 Description

Let us consider a motivating scenario in which a company built a device called ”Fuel optimiser” in charge of reducing the fuel consumed by vehicles (car, boat, tractor, lorry, etc). To improve the quality of their products, they analyse metrics (fuel consumption per km) collected from vehicles. At the end of each trip, the vehicle sensors sent metrics to a company application via REST messages. The application must face requirements like:

- The application must be close to vehicles (geo-diversity).
- Unpredictable and unlimited growth of vehicles.
- Peaks and unpredictable workloads.

To address these challenges, the architecture of this application and the infrastructure need to be flexible, highly available, well performing, reliable and scalable. The application uses a three-tier model; the vehicle sensors are directly connected to the **frontend** tier, the **middle** tier analyses the metrics collected, and the **storage** tier stores the metrics into a database. The application described in this scenario is used for the evaluation of soCloud elasticity, and high availability.

[9] http://socloud.soceda.cloudbees.net

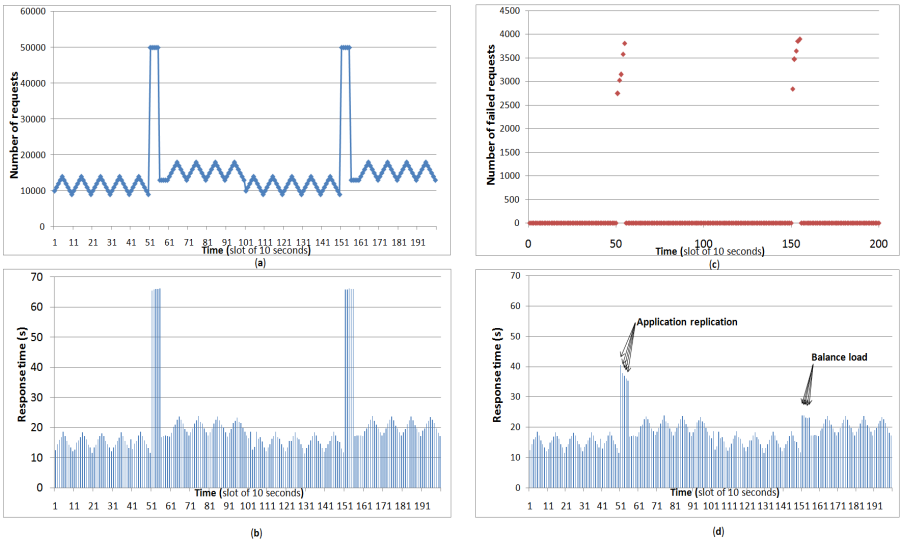


Fig. 6 The series of two flash crowd effects. (a) Effective number of requests during the evolution of the scenario. (b) Response time experienced by clients during the flash crowd effect without soCloud elasticity. (c) Number of requests failed during the two phases of the flash crowd effect. (d) Response time experienced by clients during the flash crowd effect with soCloud elasticity.

4.2 Measure of the reaction time to flash crowd effects

We have implemented a prototype of the application described in Section 5.1 and deployed it with the soCloud platform. Our soCloud platform is deployed on ten different cloud providers. We have conducted an experimental evaluation of this application in order to assess how the soCloud platform behaves and effectiveness at sustaining flash crowd effects[10]. We conducted an analysis of: (a) the application is deployed without the soCloud elasticity mechanism, and (b) the application is deployed with the soCloud elasticity mechanism.

4.2.1 Without soCloud elasticity

In the first case, we have observed the behavior of this application without elasticity capability under high request load. Each request triggers an operation that consists of analysing metrics collected by a vehicle and stores the results into a database. To that end, we have configured httperf [32] to create 50,000 connections, with 10 requests per connection and a number of new connections created per second varying between 10 and 150 ; this corresponds to a total of 3,020,000 requests. Fig. 6**(a)** shows the number of requests achieved by the application with two phases of a flash crowd effect, and Fig. 6**(b)** shows the corresponding response time (computed as the number of operations performed). During the two phases of the flash crowd effect, the average response time is 65.90 seconds. Fig. 6**(a)** and 6**(b)** show a mounted sudden load caused by the flash crowd effect. We have noted that the number of requests

[10] The flash crowd effect, also called the slash dot effect, resulte from a sudden increase in request traffic.

increases with the response time. Then, Fig. 6**(c)** shows the number of request errors, and shows the corresponding number of the failed requests. Thus, during the flash crowd effect, 1.13% of requests have failed, precisely 34,039 requests. These request errors are due to the processing timeout that we have set at 5 seconds for each request. In fact, this timeout means that the lack of any server activity on the TCP connection for this duration will be considered to be an error.

Overall, when the application becomes saturated, it suffers from performance failures and cause long response delays. We observe that the application can sustain the request rate only up to a certain limit, which directly depends on the number of requests on a time interval.

4.2.2 With soCloud elasticity

In the second case, we have studied the evolution of the response time during the two phases of the flash crowd effect when soCloud elasticity is activated.

We assume that resource(VM) is preallocated and a soCloud agent is deployed inside. Fig. 6**(d)** shows the results of the same experiment when using the soCloud elasticity mechanism. We initially observe some contention at the source of application as the response time decreases. During the first phase of the flash crowd effect, the average response time is 37.30 seconds. Indeed, the soCloud platform has detected peak mounted in *300 ms*. After 4 seconds, the soCloud platform replicates the application into another soCloud agent and updates the load balancer table for balancing charge across different instances of the application. This reaction appears clearly in Fig. 6**(d)**, where the application replication is performed. The soCloud load balancer dispatches the requests among the two instances of the application and the response time remains small despite the high traffic. During the second phase of the flash crowd effect, the application was already deployed, the soCloud platform has detected peak mounted in *300 ms*. As shown in Fig. 6**(d)**, we do not notice mounted peak during the second phase of the flash crowd effect, and the average response time is 23.38 seconds. The relatively small response time during the second phase of the flash crowd is due to the fact that the soCloud platform has already replicated the application.

Overall, at the peak of the flash crowd, all the requests are performed with zero failure and relatively acceptable response time, the soCloud platform allows the application to scale more with better quality of service. These results demonstrate that the soCloud platform deals well with elasticity across multiple cloud providers.

4.3 soCloud behavior against failures

We perform all our evaluation with the application described in the previous sections. To show the behavior in soCloud over time as failures are injected, we deploy soCloud as described in Fig. 4. The deployment of soCloud is done on ten clouds. The soCloud master is replicated to tolerate more faults. The leader and follower of the soCloud master are deployed respectively on dotCloud and CloudBees. soCloud agents are deployed on Amazon EC2, Windows Azure, DELL KACE, OpenShift, Jelastic, Heroku, Appfog, and our Eucalyptus private cloud.

4.3.1 Deployment time of soCloud

As described in Section 3, the deployment of soCloud consists of the deployment of both a soCloud master and several agents.

soCloud master The deployment of a soCloud master is done by deploying both leader and follower instances on two different clouds to ensure the high availability. The deployment on each cloud consists of deploying the execution environment (FraSCAti) with the soCloud master. The deployment of a soCloud master takes about *2.1 minutes*.

soCloud agent We measure the time for the deployment of one soCloud agent. The deployment consists of deploying the execution environment (FraSCAti) with the soCloud agent. The deployment of a soCloud agent takes about *0.9 minute*.

Overall, the average time taken to deploy soCloud with two masters (leader and follower) and one agent is about **3 minutes**.

4.3.2 Failure and recovery of a soCloud master leader

We assume that soCloud is running and our scenario application is deployed. To simulate a failure, we stop the soCloud master leader in dotCloud. In our observations, soCloud takes about *3.5 minutes* average to recovery and to become operational. soCloud takes less than 200 ms to elect a new leader. The recovery process is performed as follows. First, the soCloud master follower becomes leader after the election and rollbacks the system. Then, a new soCloud master follower is deployed on another cloud. Finally, the soCloud agent discovers automatically the new soCloud master leader. According to [30, 1], the average Mean Time To Recovery (MTTR) for public clouds is **7.5 hours**. As a comparison, the recovery time of soCloud takes only **3.5 minutes** as shown in Table 1.

Table 1 MTTR results

	MTTR(Hour)
soCloud	0.06 hour
Public clouds	7.5 hours

Failure and recovery of a soCloud master follower In this case, we simulate the failure of a soCloud master follower in CloudBees, the soCloud master leader detects automatically the failure. The soCloud master leader takes about *1200 ms* to elect and start a new master follower.

Downtime of an application deployed on soCloud The failure of an application deployed with soCloud does not affect its availability. In fact, when a failure occurs, the load balancer takes about *300* ms to detect and switch automatically to another instance of the application deployed.

Downtime of a soCloud agent The failure of a single soCloud agent does not affect the availability of the application deployed on soCloud. The soCloud load balancer allows to redirect the requests to another instance of the application. The soCloud agent deployment and start still take about *0.9 minute*. However, the deployment time of applications that were on the platform depends on the size of these applications.

4.3.3 The soCloud high availability

Let us consider the availability equation below [27,39]:

$$Availability = \frac{MTBF}{MTBF + MTTR} \quad (5)$$

As Equation 5 shows, the longer the MTTR is, the worse off a system is. The formula illustrates how both Mean Time Between Failure (MTBF) and MTTR impact the overall availability of a system. As MTTR goes up, availability goes down. To compare the availability of soCloud and public clouds, we must estimate the same MTBF. Then, in a year we assume that the MTBF is 8760 hours. The availability is calculated in Table 2.

Table 2 Availability comparison

	Availability
soCloud	$\frac{8760}{8760+0.06} = 99.999\%$
Public clouds	$\frac{8760}{8760+7.5} = 99.914\%$

Overall, as shown in Table 2, the availability of public clouds is 99.914%. As a comparison, the soCloud availability is 99.999%. This result is close from the expected reliability of mission critical systems (c.f. Section 2.4). The soCloud platform increases high availability. This result demonstrates that soCloud ensures well high availability across multiple clouds.

4.4 Overhead introduced by soCloud

In order to analyse the overhead introduced by the soCloud platform, we have deployed our use case application directly on CloudBees and through the soCloud platform. We have packaged two different archive files. The first archive file is a WAR file, its size is 50.7 Mb. This file contains the application and the execution environment FraSCAti. The second archive file is a Zip file (an SCA contribution), its size is 2.1 Mb. The second file contains only the application. The WAR and Zip files are deployed respectively on CloudBees and soCloud. The deployment of the WAR and Zip files is performed ten times. Table 3 reports the average deployment time of each file.

Table 3 Deployment time of the Zip and WAR files

Implementation	**File size**	**Avg. deploy. time**
Zip File (Application)	2.1 Mb	5301.5 ms
WAR File (Application + FraSCAti)	50.7 Mb	80830.8 ms

As noticed, the deployment time of the application directly on CloudBees is greater than the deployment time on soCloud. This is explained by the size of the WAR file which is greater than the Zip file. In fact, uploading a small file in the network is faster than a big file. When deployed the Zip file on the soCloud platform, the execution environment is already deployed and started. This is not the case of the

WAR file which contains the FraSCAti execution environment that will be installed and instantiated on the CloudBees PaaS before deploying the application on it.

To evaluate the overhead introduced by the soCloud platform, $10,000$ requests were generated and sent with the Httperf tool. We evaluate two implementations of this scenario: i) the application without soCloud, and ii) with soCloud. The scenario was executed ten times on each of the two implementations. Table 4 presents the results of the average execution time for each implementation, as well as the mean overhead introduced by the soCloud platform.

Table 4 Execution time and Overhead

Implementation	Avg. exec. time	soCloud overhead
(Application + FraSCAti)	10.85 sec	-
(Application + FraSCAti + soCloud)	11.10 sec	2.3%

From the results presented in Table 4, we can notice that the overhead introduced by the soCloud platform is 2.3%. This overhead is generated by the soCloud monitoring and the Load Balancing components. The overhead of the monitoring component is due to the information collected for the elasticity.

Overall, the abstraction provided by the soCloud platform is not free, because it introduces an overhead of 2.3%. However, the benefits provided by the soCloud platform in multi-cloud environment outweigh the difference in the execution time.

5 Related work

Related to the Inter-Cloud Architectural taxonomy presented in [20], soCloud can be classified into the Multi-Cloud service category. This section presents some of the related work to multi-cloud computing challenges discussed in Section 2: *portability*, *provisioning*, *elasticity*, and *high availability* across multiple clouds.

Multi-cloud portability Portability approaches can be classified into three categories-[33]: *functional portability*, *data portability* and *service enhancement*. The authors[37] of mOSAIC deal with *service enhancement* portability at IaaS and PaaS levels. mOSAIC provides a component-based programming model with asynchronous communication. However, mOSAIC APIs are not standardized and are complex to put at work in practice. Our soCloud solution deals with *service enhancement* portability with an API that runs on existing PaaS and IaaS. soCloud supports both synchronous and asynchronous communications offered by the SCA standard. Moreover, SCA defines an easy way to use portable API. The Cloud4SOA [14] project deals with the portability between PaaS using a semantic approach. soCloud intends to provide portability using an API based on the SCA standard.

Multi-cloud provisioning A great deal of research on dynamic resource allocation for physical and virtual machines and clusters of virtual machines [2] exists. The work of dynamic provisioning of resource in cloud computing may be classified into two categories. Authors in [31] have addressed the problem of provisioning resources at the granularity of VMs. Other authors in [10] have considered the provisioning of resources at a finer granularity of resources. In our work, we consider provisioning at both VM and finer granularity of resources.

The authors in [17] have addressed the problem of deploying a cluster of virtual machines with given resource configurations across a set of physical machines. While [13] defines a Java API permitting developers to monitor and manage a cluster of Java VMs and to define resource allocation policies for such clusters. Unlike [17, 13], soCloud uses both an application-centric and virtual machine approaches. Using knowledge on application workload and performance goals combined with server usage, soCloud utilizes a more versatile set of automation mechanisms.

Multi-cloud elasticity Managing elasticity across multiple cloud providers is a challenging issue. However, although managed elasticity through multiple clouds would benefit when outages occur, few solutions are supporting it. For instance, in [7], the authors present a federated cloud infrastructure approach to provide elasticity for applications, however, they do not take into account elasticity management when outages occur. Another approach was proposed by [40], which managed the elasticity with both a controller and a load balancer. However, their solution does not address the management of elasticity through multiple cloud providers. The authors in [29] propose a resource manager to manage application elasticity. However, their approach is specific for a single cloud provider.

Multi-cloud high availability Cloud providers such as Amazon EC2, Windows Azure, Jelastic already provide a load balancer service with a single cloud to distribute load among virtual machines. However, they do not provide load balancing across multiple cloud providers. Different approaches of dynamic load balancing have been proposed in the literature [8,21], however, they do not provide a mechanism to scale the load balancers themselves. The authors in [38] have explored the agility way to quickly reassign resources. However, their approach does not take into account a multi cloud environment. Most existing membership protocols [4] employ a consensus algorithm to achieve agreement on the membership. Achieving consensus in an asynchronous distributed system is impossible without the use of timeouts to bound the time within which an action must take place. Even with the use of timeouts, achieving consensus can be relatively costly in the number of messages transmitted, and in the delays incurred. To avoid such costs, soCloud uses a novel Leader Determined Membership Protocol that does not involve the use of a consensus algorithm.

6 Discussion and limitations

soCloud is a PaaS to aggregate multiple clouds. Throughout the article, we have essentially discussed the advantages of soCloud. On the one hand, soCloud may miss some features that are provided by the underlying clouds used. In other words, soCloud may not exploit the specific features (i.e., elasticity rules, provisioning properties, replication trigger) that is not provided by it. On the other hand it may be the case that some developers or companies may not like to use an SCA-based approach. Indeed, the soCloud adoption can become therefore an issue. One approach for soCloud to address these concerns is to use a wrapper that enable transparent access to cloud provider features. As an SCA-based approach, soCloud offers a solution to deploy and execute service oriented applications. It would be useful in future work for soCloud to overcome the constraint of supporting only SCA-based applications.

When deployed the soCloud platform on other PaaS, the scaling mechanism of these platforms is not used by our platform in order to avoid duplicated mechanism.

The cloud platforms and their features provided, especially at the PaaS level are evolving dynamically. However, in general the problem of maintaining the mappings to various cloud providers and managing this evolution to keep up with recent features of our supported clouds are a concern. A common way to address these issues is by wrapping them as soCloud features. However, the use of the future standard for Cloud computing is still the best approach.

soCloud provides an abstraction to hide the heterogeneity and the complexity of the underlying clouds. The solution provided by soCloud can introduce an additional cost (i.e., in term of performance, footprint) to existing IaaS/PaaS environments. However, soCloud provides a uniform way to deploy, execute and manage applications in multi-cloud environments. As benefits, the developer focuses on the cloud rather than troubleshooting implementations, exploits multi-cloud portability, has an efficient management of her applications across multi-cloud. In comparison to heterogeneous ways offered by the several IaaS/PaaS solutions, soCloud provides many benefits.

7 Conclusion

In this article, we have proposed soCloud a service-oriented component-based PaaS for managing portability, provisioning, elasticity, and high availability across multiple clouds. soCloud is a distributed PaaS that provides a model for building any multi-cloud SaaS applications. This model is based on an extension of the OASIS SCA standard. We surveyed each of the concepts related to express specific elasticity rules, ensure high availability across multiple clouds and pointed out problematics. To address these problems, this article proposes an architecture, and describes the interactions between each component of this architecture. We explain how the components in a soCloud application descriptor can be annotated with elasticity rules, placement constraints, computation constraints. Based on these annotations, deployable contributions can be loaded and deployed in a suitable manner. The article described the approach used by the soCloud platform to ensure high availability. In particular soCloud takes a wait-free approach to the problem of coordinating components in different clouds and uses load balancer to switch from one application instance to another in case of failures. In comparison, the soCloud's availability with public [30] cloud availability, we demonstrate that soCloud ensures high availability in minutes instead of hours. We analyse the flash crowd phenomenon on a use case, and demonstrate how the soCloud platform increases the elasticity of the application. This approach is proactive in the case that the content replication is performed when detecting a traffic surge and anticipating a flash crowd.

As of future work, we plan to continue our research in the following directions. First, currently, soCloud manages application's components as contribution file in terms of packaging and deployment. The archive that is referred to by *implementation.contribution* may be an artifact within a larger contribution (i.e., an EAR or WAR file inside a larger ZIP file), or archive may itself be a contribution. Indeed, soCloud will manage and deploy all Java EE archive (WAR, EAR). Second, we will

investigate how the concept of aggregated multiple clouds can be used to reduce the resource provisioning cost, while maintaining the Quality of Service (QoS) to customers who use the resources. Third, as many organizations need to move data from one cloud to another we will work on data portability in a multi-cloud environment.

8 Acknowledgment

This work is partially funded by the ANR (French National Research Agency) ARPEGE SocEDA project and the EU FP7 PaaSage project.

References

1. Lessons Learned from Recent Cloud Outages (2013). http://tinyurl.com/qz5maey
2. Anedda, P., Leo, S., Manca, S., Gaggero, M., Zanetti, G.: Suspending, Migrating and Resuming HPC virtual clusters. Future Generation Computer Systems **26**(8), 1063–1072 (2010)
3. Armbrust, M. and Fox, A. Griffith, R. Joseph, A.D. Katz, R. Konwinski, A. Lee, G. Patterson, D. Rabkin, A. Stoica, I. et al.: A view of cloud computing. Communications of the ACM **53**(4), 50–58 (2010)
4. Birman, K.P., Van Renesse, R., et al.: Reliable distributed computing with the Isis toolkit, vol. 85. IEEE Computer Society Press Los Alamitos (1994)
5. Bouteiller, A., Lemarinier, P., Krawezik, K., Capello, F.: Coordinated checkpoint versus message log for fault tolerant mpi. In: Cluster Computing, 2003. Proceedings. 2003 IEEE International Conference on, pp. 242–250. IEEE (2003)
6. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The Fractal component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. Softw. Pract. Exper. **36**(11-12), 1257–1284 (2006)
7. Buyya, R., Ranjan, R., Calheiros, R.: Intercloud: Utility-oriented federation of cloud computing environments for scaling of application services. Algorithms and architectures for parallel processing pp. 13–31 (2010)
8. Cardellini, V., Colajanni, M., Yu, P.: Dynamic load balancing on web-server systems. Internet Computing, IEEE **3**(3), 28–39 (1999)
9. Chang, E., Roberts, R.: An improved algorithm for decentralized extrema-finding in circular configurations of processes. Communications of the ACM **22**(5), 281–283 (1979)
10. Chase, J.S., Anderson, D.C., Thakar, P.N., Vahdat, A.M., Doyle, R.P.: Managing energy and server resources in hosting centers. In: ACM SIGOPS Operating Systems Review, vol. 35, pp. 103–116. ACM (2001)
11. Chen, Z., Liu, Z., Stolz, V., Yang, L., Ravn, A.P.: A refinement driven component-based design. In: Engineering Complex Computer Systems, 2007. 12th IEEE International Conference on, pp. 277–289. IEEE (2007)
12. Coulouris, G., Dollimore, J., Kindberg, T.: Distributed systems: concepts and design. Addison-Wesley Longman (2005)
13. Czajkowski, G., Wegiel, M., Daynes, L., Palacz, K., Jordan, M., Skinner, G., Bryce, C.: Resource management for clusters of virtual machines. In: Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium on, vol. 1, pp. 382–389. IEEE (2005)
14. Dandria, F., Bocconi, S., Cruz, J.G., Ahtes, J., Zeginis, D.: Cloud4SOA: Multi-Cloud Application Management Across PaaS Offerings. In: Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), 2012 14th International Symposium on, pp. 407–414. IEEE (2012)
15. Erl, T.: Soa: principles of service design, vol. 1. Prentice Hall Upper Saddle River (2008)
16. Etzion, O., Niblett, P.: Event Processing in Action. Manning Publications Co. (2010)
17. Foster, I., Freeman, T., Keahy, K., Scheftner, D., Sotomayer, B., Zhang, X.: Virtual clusters for grid communities. In: Cluster Computing and the Grid, 2006. CCGRID 06. Sixth IEEE International Symposium on, vol. 1, pp. 513–520. IEEE (2006)
18. Garg, V.K.: Concurrent and distributed computing in Java. Wiley-IEEE Press (2005)

19. Gonzalez, H., Halevy, A.Y., Jensen, C.S., Langen, A., Madhavan, J., Shapley, R., Shen, W., Goldberg-Kidon, J.: Google fusion tables: web-centered data management and collaboration. In: Proceedings of the 2010 international conference on Management of data, pp. 1061–1066. ACM (2010)
20. Grozev, N., Buyya, R.: Inter-Cloud Architectures and Application Brokering: Taxonomy and Survey. Software: Practice and Experience (2012). DOI 10.1002/spe.2168. http://dx.doi.org/10.1002/spe.2168
21. Harchol-Balter, M., Downey, A.: Exploiting process lifetime distributions for dynamic load balancing. ACM Transactions on Computer Systems (TOCS) **15**(3), 253–285 (1997)
22. InfoWorld: The 10 worst cloud outages (and what we can learn from them). http://tinyurl.com/br9ck4a
23. Isard, M.: Autopilot: automatic data center management. ACM SIGOPS Operating Systems Review **41**(2), 60–67 (2007)
24. Karn, P., Partridge, C.: Improving round-trip time estimates in reliable transport protocols. ACM SIGCOMM Computer Communication Review **17**(5), 2–7 (1987)
25. Luckham, D., Schulte, R.: Event Processing Glossary - Version 1.1. Processing **1.1**(July), 1–19 (2008). http://complexevents.com/wp-content/uploads/2008/08/epts-glossary-v11.pdf
26. Malpani, N., Welch, J.L., Vaidya, N.: Leader election algorithms for mobile ad hoc networks. In: Proceedings of the 4th international workshop on Discrete algorithms and methods for mobile computing and communications, pp. 96–103. ACM (2000)
27. Marcus, E., Stern, H.: Blueprints for high availability. Wiley (2003)
28. Marino, J., Rowley, M.: Understanding SCA (Service Component Architecture). Addison-Wesley Professional (2010)
29. Marshall, P., Keahey, K., Freeman, T.: Elastic site: Using clouds to elastically extend site resources. In: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 43–52. IEEE Computer Society (2010)
30. Maurice Gagnaire, Felipe Diaz,Camille Coti, Christophe Cerin, Kazuhiko Shiozaki, Yingjie Xu, Pierre Delort, Jean-Paul Smets, Jonathan Le Lous, Stephen Lubiarz, Pierrick Leclerc: Downtime statistics of current cloud solutions (2012)
31. Mietzner, R., Leymann, F.: Towards provisioning the cloud: On the usage of multi-granularity flows and services to realize a unified provisioning infrastructure for saas applications. In: Services-Part I, 2008. IEEE Congress on, pp. 3–10. IEEE (2008)
32. Mosberger, D., Jin, T.: httperf a tool for measuring web server performance. SIGMETRICS Perform. Eval. Rev. **26**(3), 31–37 (1998). DOI 10.1145/306225.306235. Http://doi.acm.org/10.1145/306225.306235
33. Oberle, K., Fisher, M.: ETSI CLOUD–initial standardization requirements for cloud services. In: Economics of Grids, Clouds, Systems, and Services, pp. 105–115. Springer (2010)
34. Paraiso, F., Haderer, N., Merle, P., Rouvoy, R., Seinturier, L.: A Federated Multi-Cloud PaaS Infrastructure. In: 5th IEEE International Conference on Cloud Computing, pp. 392 – 399. Hawaii, United State (2012). DOI 10.1109/CLOUD.2012.79. http://hal.inria.fr/hal-00694700
35. Paraiso, F., Hermosillo, G., Rouvoy, R., Merle, P., Seinturier, L.: A Middleware Platform to Federate Complex Event Processing. In: Sixteenth IEEE International EDOC Conference, pp. 113–122. Springer, Beijing, China (2012). http://hal.inria.fr/hal-00700883
36. Paraiso, F., Merle, P., Seinturier, L.: Managing Elasticity Across Multiple Cloud Providers. In: 1st International workshop on multi-cloud applications and federated clouds. Prague, Czech, Republic (2013). http://hal.inria.fr/hal-00790455
37. Petcu, D., Macariu, G., Panica, S., Crăciun, C.: Portable Cloud applications From theory to practice. Future Generation Computer Systems (2012)
38. Qian, H., Miller, E., Zhang, W., Rabinovich, M., Wills, C.E.: Agility in virtualized utility computing. In: Virtualization Technology in Distributed Computing (VTDC), 2007 Second International Workshop on, pp. 1–8. IEEE (2007)
39. Torell, W., Avelar, V.: Mean time between failure: Explanation and standards. White Paper **78** (2004)
40. Vaquero, L., Rodero-Merino, L., Buyya, R.: Dynamically scaling applications in the cloud. ACM SIGCOMM Computer Communication Review **41**(1), 45–52 (2011)
41. Wang, Y.M.: Consistent global checkpoints that contain a given set of local checkpoints. Computers, IEEE Transactions on **46**(4), 456–468 (1997)
42. Zdnet: Amazon cloud down; Reddit, Github, other major sites affected (2012). http://tinyurl.com/95kmk8y
43. Zhang, Q., Cheng, L., Boutaba, R.: Cloud computing: state-of-the-art and research challenges. Journal of Internet Services and Applications **1**(1), 7–18 (2010)