Walraven *et al. Journal of Internet Services and Applications* (2015) 6:2
DOI 10.1186/s13174-014-0017-x

Journal of Internet Services and Applications
a SpringerOpen Journal

RESEARCH Open Access

PaaSHopper: Policy-driven middleware for multi-PaaS environments

Stefan Walraven*†, Dimitri Van Landuyt†, Ansar Rafique†, Bert Lagaisse† and Wouter Joosen†

Abstract

Offering Software-as-a-Service (SaaS) applications on top of a Platform-as-a-Service (PaaS) platform is a promising strategy as the SaaS provider does not need to acquire and maintain private cloud infrastructure, and it enables him/her to enjoy the benefits of cloud scalability and flexiblity as well. However, as this entails losing some control over the application and its data, SaaS providers are in practice reluctant to migrate to a PaaS platform entirely. To alleviate such concerns of vendor lock-in, the concept of a multi-cloud involves integrating and combining multiple cloud environments, private as well as public, but also involving multiple providers and different technologies. This has the added benefit that it further improves overall availability, flexibility and scalability. Current support for multi-cloud applications however is limited.

This paper presents PaaSHopper, a middleware platform for developing and operating multi-tenant SaaS applications in a multi-PaaS environment. It enables the SaaS provider to have fine-grained control over the execution of applications and the storage of application data, while offering the tenant some degrees of customization and self-service as well. Driven by stakeholder-specific policies, the middleware dynamically decides which requests and tasks are executed in a particular part of the multi-PaaS environment. We validated this work in the context of four realistic SaaS application cases on top of a multi-cloud consisting of a local JBoss Application Server cluster, Google App Engine, and Red Hat OpenShift.

Keywords: Multi-cloud; PaaS; Policy-driven adaptation; Middleware; Portability

1 Introduction

Cloud computing enables the on-demand delivery of ICT solutions as online services, covering software applications, system software, and hardware infrastructure [1-3]. High flexibility and scalability benefits are gained by allowing these services to be provisioned rapidly upon customer request. The cloud computing paradigm includes three cloud service delivery models [1,3]: (i) Infrastructure as a Service (IaaS), for example Amazon EC2 [4], delivers fundamental computing resources, such as processing, storage and network capacity, as a service, (ii) Platform as a Service (PaaS) provides a higher-level application development and hosting platform, for example Google App Engine (GAE) [5], and (iii) Software as a Service (SaaS) delivers software applications as online, on-demand services, e.g. Salesforce CRM [6].

In this paper, we focus on the PaaS delivery model. PaaS is a promising development and deployment platform for enterprise SaaS applications [7,8], as it allows the SaaS provider him- or herself to enjoy the cloud benefits of high availability, on-demand scalability, and pay-per-use cost models without having to acquire and manage the underpinning cloud infrastructure.

However, several concerns still withhold the general adoption of this strategy [9-11]. By building SaaS applications on top of a PaaS platform, the SaaS provider partially loses control over his/her applications and data. Especially for core business applications, many SaaS providers prefer their own private cloud or data center, but this requires large investments and has a capacity that is limited in practice. In addition, depending on a single PaaS provider comes with the non-negligible risks of provider and technology lock-in as well as limited availability [12-14]. In addition, SaaS providers typically adopt a multi-tenant architecture [15,16] to achieve economies of scale: a single application instance is shared by many different customer

*Correspondence: stefan.walraven@cs.kuleuven.be
†Equal Contributors
iMinds-DistriNet, KU Leuven, Celestijnenlaan 200A, 3001 Leuven, Belgium

organizations (tenants), each in turn servicing their own end users. High operational cost efficiency is achieved by sharing the same resources among multiple customer organizations. In practice however, multi-tenancy requires even more flexibility and scalability to address the fluctuating number of (active) tenants and to support all variations of the respective tenant requirements at once.

To address these disadvantages, there is a growing interest in multi-cloud solutions [17]. We define a *multi-cloud* as a composition of multiple cloud environments with as purpose to improve availability and flexibility, and to avoid vendor lock-in. A hybrid cloud is a multi-cloud that consists of at least a public and a private cloud environment, thus combining the unlimited capacity of public clouds with the increased control of private clouds into an integrated system. However, current support for multi-cloud applications is still fairly limited.

In this paper, we focus on the key challenges faced by the SaaS provider when deploying SaaS applications in a multi-PaaS environment:

1. *Heterogeneity in development and deployment platform:* In essence, the current PaaS platforms offer a lot of similar architectural concepts towards application developers, but typically via vendor-specific solutions using different programming languages and supporting technologies [7]. This heterogeneity hinders portability and interoperability of SaaS applications across different PaaS platforms.
2. *Flexible and (re)configurable decisions that need to be supported:* There are large differences in the trust relationship between the customers and the different (public) cloud providers. This requires smart and fine-grained control (up to the tenant level) over the execution of multi-tenant applications and storage of data across multi-PaaS environments.

This paper presents PaaSHopper, a policy-driven middleware that addresses the above-mentioned challenges by (i) offering a PaaS abstraction layer to increase portability and interoperability of application components over different PaaS platforms and providers, and (ii) facilitating flexible, reconfigurable deployment and execution, driven by policies that express stakeholder-specific constraints such as geographical location, security, and workload.

A first version of the PaaSHopper middleware architecture has been presented in previous work [18,19]. In contrast, this paper introduces an extensive motivation based on four different SaaS application cases, all based on our analysis of industrial SaaS applications. In addition, the middleware architecture has been extended and the validation strengthened. This paper also provides an extensive discussion of the PaaSHopper middleware and the remaining open challenges.

The remainder of this paper is structured as follows. Section 2 introduces several multi-cloud scenarios from four realistic SaaS application cases, and identifies the challenges that are addressed in this paper. Section 3 elaborates on the architecture of the PaaSHopper middleware with respect to portability, interoperability, and policy-driven execution and storage. Section 4 validates the middleware based on a prototype implementation. In Section 5, we discuss the work and identify the open challenges ahead. Section 6 discusses related work and Section 7 concludes the paper.

2 Motivation

The motivation for this paper is based on our experience with a number of multi-cloud applications, obtained in the context of several applied research projects in collaboration with industrial SaaS providers. In Section 2.1, we discuss a set of realistic cases of SaaS applications in multi-cloud environments, from which we derive in Section 2.2 the key challenges for multi-cloud applications.

2.1 Multi-cloud application cases

The following cases present a number of the deployment and operation aspects of four multi-tenant SaaS applications, while illustrating the benefits of multi-cloud environments. Each of these applications have different properties and requirements with respect to execution and storage.

Application #1: Document processing as a service. This multi-tenant SaaS application delivers B2B document processing facilities to a wide range of companies (see also [20]). It supports the business-specific generation, the archival and the delivery of large sets of customized digital documents. This SaaS application is deployed on top of a hybrid cloud solution, consisting of a private cloud platform that is managed by the SaaS provider, and a public cloud offering that is used as a spill-over to address peaks in the processing load. The storage of the documents occurs at the same location in the hybrid cloud as the processing.

However, the various types of data and documents (e.g. invoices, payslips, medical reports and leaflets) have different requirements with respect to confidentiality. For example, invoices may only be processed and stored in a cloud environment where certain security requirements are guaranteed (encrypted communication and storage), while there are no such constraints for generating leaflets. In addition, the SaaS provider aims to maximally utilize his/her on-premise infrastructure.

Application #2: Log management as a service. This B2B cloud offering integrates with the on-premise infrastructure of the different tenants: a local agent collects and aggregates the logs of the applications and infrastructure, and sends them to the log management service. This service performs complex analysis activities on the collected logs (e.g. detection of suspicious activities) and heavily relies on scalable storage. To ensure the necessary availability and scalability, the SaaS provider deploys the application in a multi-cloud environment consisting of a number of geographically distributed private data centers.

As the log management service is a data-driven application, the analysis activities should occur near the storage location to avoid the (expensive) migration of large data sets. In addition, tenant-specific constraints are applicable to the geographical placement of the data. For example, a financial company requires that the data may only be stored in a data center in the same country, or even using a dedicated storage infrastructure to ensure strict isolation.

Application #3: Medical image processing as a service. In this application, medical images from different hospitals are processed and stored online as part of the electronic health record (EHR). The SaaS provider uses a multi-cloud solution to distribute and replicate the data over multiple data centers. These data centers can be managed by external (certified) companies, but are in practice not part of a public cloud offering.

Typically, medical images are large files and subject to strict rules with respect to privacy. As a consequence, different hospitals have different requirements with respect to the processing and storage of these medical images, especially driven by governmental rules. For example, European medical data should be stored within Europe, or even more strict, a specific tenant can require that the data may not be stored in a data center that is hosted by a US company, even if it is located in Europe.

Application #4: Simulation processing as a service. This enterprise SaaS offering provides services to perform simulations and optimizations of engineering processes for companies in the automotive and aerospace industry. After the simulation process, the results should be presented to the respective tenants. The amount of data sent throughout this application is limited (e.g. input parameters and end results), but the simulations and optimizations are CPU-intensive. Therefore, a hybrid cloud solution is used to outsource the processing to the public cloud.

However, the input data as well as the simulation results can be highly confidential, for example information about new prototypes, thus putting restrictions on the storage. In this case, processing could be allowed in the public cloud, but the results should be stored in the private cloud. Similar to the first application, the SaaS provider also aims to maximally use his/her on-premise infrastructure for processing before doing a spill-over.

2.2 Challenges

Based on the application cases presented above, we have identified the following challenges to support the development and deployment of SaaS applications in multi-PaaS environments:

- *Portability and interoperability across different PaaS platforms:* There is a large variety in PaaS platforms, possibly supporting different programming languages and technologies. Even when only considering Java-based platforms, differences exist in terms of development API and deployment: each platform offers its own vendor-specific solution for interfacing with the platform itself as well as with its cloud services, such as scalable storage and background workers (for asynchronous execution) [7,21,22]. Moreover, not every platform supports the same cloud services. This heterogeneity hinders the portability and interoperability of SaaS applications across different PaaS platforms, and thus also the deployment in multi-PaaS environments.
- *Fine-grained control over execution and storage:* Tenants can impose constraints concerning the geographical location of the processing and/or storage of their data, security, available cloud services, etc. Furthermore, SaaS providers want to ensure availability and address peak loads, but also want to limit the use of (more expensive) external cloud infrastructure. Therefore, there should be support to manage and enforce these co-existing stakeholder-specific requirements within the shared SaaS application and across different cloud environments. This enforcement should not only occur for incoming requests and data, but potentially for any interaction between the different application components as well as cloud services.

3 Policy-driven middleware for multi-cloud platforms

This section presents *PaaSHopper*, a policy-driven middleware framework that enables SaaS providers as well as individual tenants to have fine-grained control over the execution and operation of the multi-tenant SaaS application in dynamic multi-PaaS environments. The PaaSHopper middleware consists of two subsystems (see Figure 1) to address the respective challenges in Section 2.2: (i) an *abstraction layer* to tackle the portability and interoperability requirements, and (ii) a *policy-driven execution layer* to control the execution and storage. The constraints and rules of the different stakeholders are

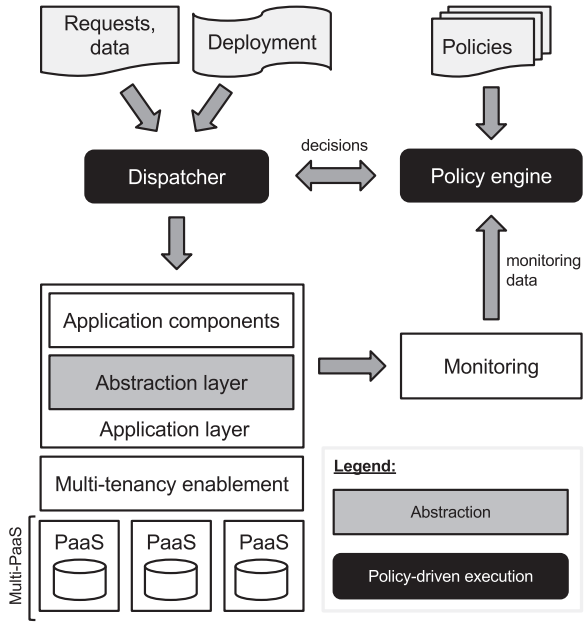

Figure 1 Concept diagram. High-level overview of the policy-driven mechanism of the proposed PaaSHopper middleware.

specified in *policies*. Driven by these different co-existing policies and in collaboration with the policy engine, a dispatcher selects at run time the appropriate application components, which are distributed over multiple (heterogeneous) PaaS offerings, to process requests and data (cf. Figure 1).

Furthermore, a multi-tenancy enablement layer (see Figures 1 and 2) offers basic *multi-tenancy* support by managing the current tenant context and by facilitating the tenant-aware isolation of application data, configurations and policies. Some PaaS platforms already offer built-in support for tenant-aware data isolation, e.g. Google App Engine (GAE) [5]. In addition, the PaaSHopper middleware ensures that the specific tenant context is passed with every invocation throughout the distributed application. This way, the middleware platform offers built-in support for creating multi-tenant applications.

The next subsections elaborate on the architecture of the middleware framework (see Figure 2), providing more details on the common abstractions offered by the abstraction layer, the cross-cloud interaction managed by the dispatcher, and the policy-driven execution. This is an open and versatile architecture, as it supports different implementations and deployments, depending on the application type and the specific nature of the multi-cloud environment.

3.1 Common abstraction for PaaS platforms

The abstraction layer (see (a) and (b) in Figure 2) is responsible for application portability across the heterogeneous multi-PaaS environment. The core of this layer offers a uniform API to the application components for interaction with the PaaSHopper middleware as well as with the underpinning PaaS platform(s). More specifically, the `AbstractPaaSPlatform` component represents the application container of the execution environment, while the `PaaSService` components provide interfaces to each of the supported cloud services (e.g. scalable storage).

An application component interacts with `AbstractPaaSPlatform` to retrieve references to these `PaaSServices` or other application components. A `CallContext` object is associated to each invocation. It contains all relevant information regarding the user and the current tenant (see Figure 3). The passing of a `CallContext` is imposed by the distributed nature of the middleware: there is no single application run-time environment or single main memory in a multi-cloud where information about the current active tenant can be stored. Explicitly passing `CallContext` simplifies the design and implementation and enables us to keep the different services stateless. Furthermore, the usage of this `CallContext` object in combination of web service standards (e.g. SOAP or REST) ensures interoperability between application components across different PaaS platforms.

For each of the different PaaS platforms, portability drivers are required to provide an implementation for the common abstraction. These drivers ensure the correct mapping to the vendor-specific APIs, for example in the form of a data access middleware for storage services. Optionally, a driver can provide a full implementation of a PaaS service that is not natively supported by the underpinning platform. Obviously, the appropriate drivers have to be deployed together with the implementation of the application to ensure proper execution.

In [19], we have defined and evaluated such a uniform API for three common PaaS services, including structured storage (NoSQL), blob storage, and asynchronous task execution. However, this solution can easily be interchanged by creating or configuring different drivers with other existing abstractions, for example Hibernate OGM [23] or Impetus Kundera [24] as data access middleware.

3.2 Cross-cloud interaction

The `Dispatcher` (see (c) in Figure 2) ensures the transparent interaction between the different application components that are distributed over the multi-PaaS environment. `AbstractPaaSPlatform` relies on the `Dispatcher` to select the appropriate component instance. However, in order to select an instance of a particular component and to interact with it, the dispatcher

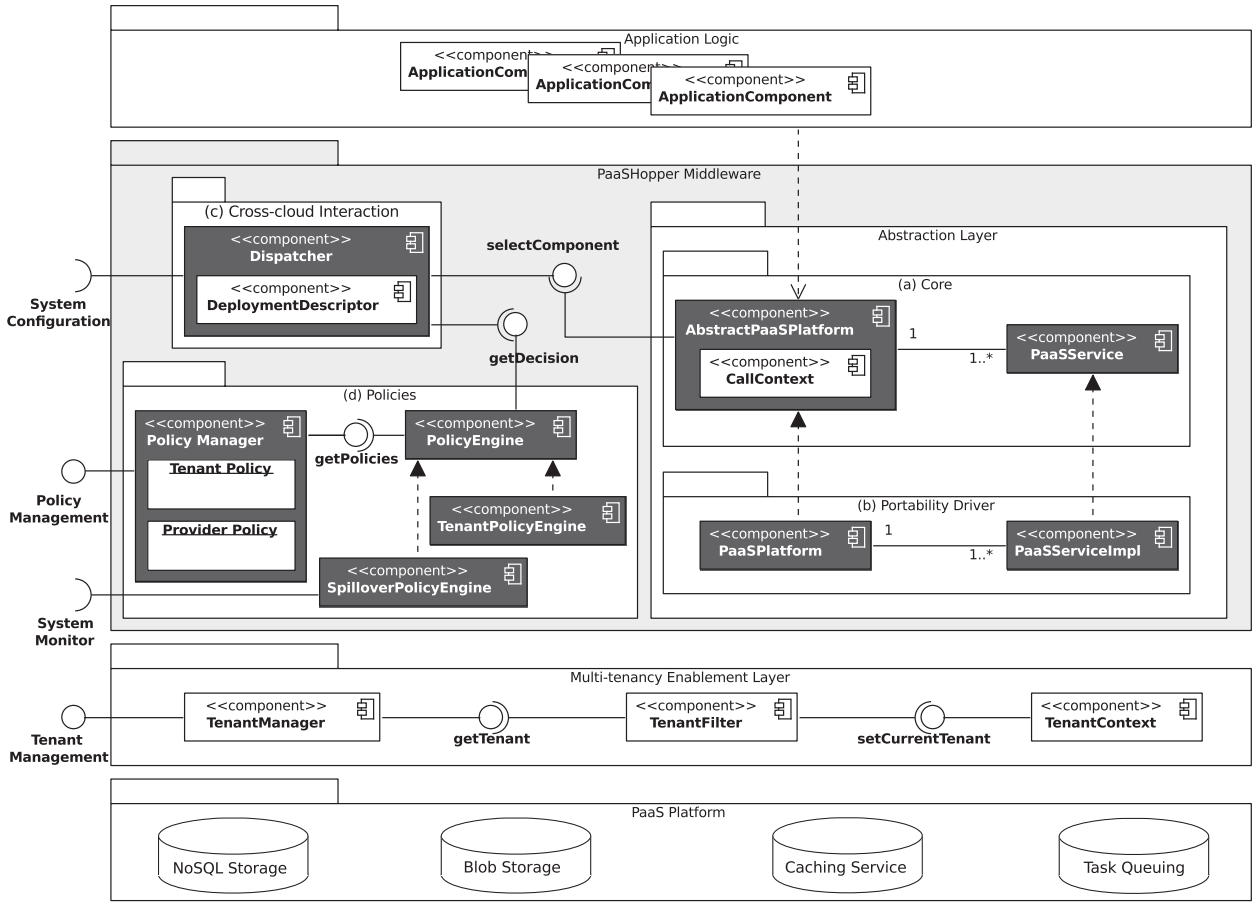

Figure 2 PaaSHopper architecture. Overview of the PaaSHopper middleware with **(a)** the uniform API, **(b)** the portability drivers, **(c)** the dispatcher, and **(d)** the policy engine.

requires an up-to-date overview of the deployment of the entire application across the multi-PaaS environment (i.e. the deployment descriptor). After an instance is selected, it is returned to the application that can start invoking operations on it. The returned component instance is either a local instance or a proxy to a remote instance, as shown in Figure 4.

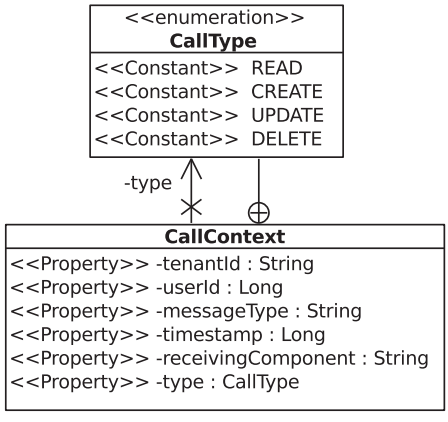

Figure 3 Call context. `CallContext` interface.

The *deployment descriptor* is a configuration artifact that specifies (i) the different PaaS platforms and their properties (e.g. private versus public cloud offering), and (ii) the deployment of all the available local and remote instances for each component interface, i.e. mapping of an instance to one of the PaaS platforms and how to access it, including metadata (e.g. description, tags). The SaaS provider can specify additional domain- and application-specific properties and corresponding values to reflect the characteristics of the different cloud platforms, for example with respect to security.

3.3 Policy-driven execution and storage

The core functionality of the PaaSHopper middleware is offering fine-grained control over the execution of a tenant's request using policies. These policies describe the constraints and rules of the different stakeholders in a declarative way, extracting them from the application code in a modular and reusable way. Based on the different

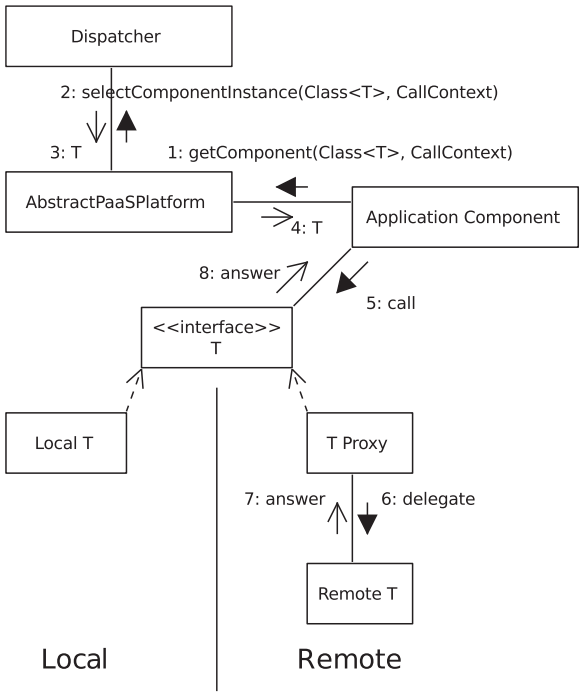

Figure 4 Interoperability. Transparent component retrieval.

tenant- and provider-specific policies, the middleware decides where in the multi-cloud a task will be executed or data will be stored. Thus, the execution flow of the multi-cloud application has to continuously adapt based on the context.

Policy types. The PaaSHopper middleware currently supports two types of policies: tenant policies and spill-over policies. A *tenant policy* defines all tenant-specific constraints. These policies are automatically isolated from each other due to our multi-tenant data store abstraction. A tenant policy is coupled to the type of the message (e.g. confidential messages) that is sent to the requested component (and which is indicated by the `messageType` field in the `CallContext`, as illustrated in Figure 3). For each message type, the tenant policy can list several constraints to which the receiving component must comply. Listing 1 defines the grammar for describing tenant policies that are currently supported.

The second policy is the *spill-over policy* and is specified by the SaaS provider. Such a policy specifies a configurable threshold for the current workload and enables the PaaSHopper middleware to dynamically decide where a task needs to be executed, either on-premise or remotely, based on the load information retrieved from the overall system monitor. When the load is low, the on-premise utilization is maximized by executing all tasks locally. When the load surpasses the configured threshold, the policy engine tries to force a remote execution.

Listing 1 Grammar of a tenant policy in BNF notation

```
<policy> ::= <component-info>,
             <message-info>,
             <constraints>
<component-info> ::=
  Component = <interface-name>
<message-info> ::=
  MessageType = <string>
  | MessageType = not <string>
<constraints> ::=
  <constraint> (',' <constraint>)*
  | (<constraints> and <constraint>)
  | (<constraints> or <constraint>)
<constraint> ::=
  <location-info>
  | <access-info>
  | <provider-info>
  | <property>
<location-info> ::=
  Location = <location>
  | Location = not <location>
<location> ::= <string> | '*'
<access-info> ::= Access = <access>
<access> ::= Public | Private | '*'
<provider-info> ::=
  Provider {
    [Company = <company-name>,]
    [HQ = <string> | not <string>,]
  }
<property> ::=
  <provider-property> = <value>
```

Policy evaluation. In the current middleware architecture (see (d) in Figure 2), the dispatcher relies on a `PolicyEngine` to select an instance of a requested component depending upon the constraints specified in the applicable policies and the context (e.g. the associated `CallContext` and the current load). The policy engine filters the set of all possible component instances that are matched by the different policies. The first component instance that complies to all imposed constraints is returned via the dispatcher to the `AbstractPaaS-Platform` component (cf. Figure 4). A unique priority integer value is assigned to each policy. This priority value is used when two policies output conflicting constraints, for example when a tenant policy requires local execution but the spill-over policy forces remote execution due to the high load. In that case the policy with the highest priority overrules the other policy. In the middleware this ensures that only tenant requests that are allowed to execute on a public cloud, are delegated when the load is high.

As multi-tenancy enables multiple tenants to use the SaaS application simultaneously, it is important to make the application components stateless and to evaluate the policies on a per call basis, i.e. at each interaction. This implies that requested instances cannot be stored within the member variables of an application component. While experimenting with the use of (traditional) dependency injection [25] techniques to inject the selected component in the client component as a member variable, it became clear to us that when executing a tenant call this often led to concurrency control issues. Dependencies thus need to be re-resolved for each call, as different

policies apply for different tenants. For example, in [26] we also required a tenant-aware dependency injector to enable tenant-specific customization of SaaS applications.

SaaS providers can also implement their own control mechanisms. The `CallContext` contains much relevant information that enables other control mechanisms to be implemented (see Figure 3). For example, an application can implement a user policy that allow to define individual policies on a per end user (of tenants) basis. Furthermore, a custom implementation of a policy engine can be provided and inserted into the dispatcher to support custom component selection.

4 Validation

Based on our analysis of the application cases presented in Section 2.1, we extracted a set of three common multi-cloud scenarios: (i) enforcing constraints of tenants and other stakeholders with respect to where data can be processed and/or stored (all applications), (ii) spill-over to the public cloud driven by provider-specific rules (cf. applications #1 and #4), and (iii) migration to other PaaS platforms to replace current providers or to expand the multi-cloud environment (all applications).

To validate the PaaSHopper middleware in the context of these scenarios, we adopted a prototype-driven approach: we implemented a prototype of the document processing application on top of the PaaSHopper middleware (Section 4.1). As the document processing application covers all these multi-cloud scenarios (Sections 4.2 till 4.4), this prototype is representative to illustrate the practical feasibility and applicability of the middleware for the four application cases.

4.1 Prototype

We have developed a Java prototype of the PaaSHopper middleware and an implementation of the document processing application, called CloudPost. This CloudPost implementation consists of a set of services that are executed in a workflow, such as a templating service, a PDF rendering service and a delivery service.

As underpinning multi-PaaS environment, we used (i) a local JBoss AS 7 cluster with a MongoDB database (representing the private cloud), (ii) Google App Engine (GAE) [5] with its datastore, and (iii) Red Hat OpenShift [27] using a Apache Tomcat 7 gear extended with a MongoDB gear for storage. As part of the abstraction layer of the PaaSHopper middleware, we implemented portability drivers for each of these platforms, offering multi-cloud support for three common PaaS services (i.e. structured storage, blob storage and asynchronous task execution). The abstraction layer is deployed on each PaaS platform, combined with the appropriate portability drivers, in order to support the deployment of the CloudPost application on top of this heterogeneous multi-cloud. As the public PaaS offerings are not necessarily used (only during high load), the entry point of the application as well as the dispatcher are deployed in the private cloud only.

Furthermore, the CloudPost provider has to define the multi-cloud environment in the deployment descriptor (see Listing 2). The PaaSHopper middleware allows the provider to specify the properties of the different PaaS platforms. For example, the private cloud provides secure communication (lines 12–13 in Listing 2), OpenShift offers both secure communication and encrypted storage (lines 38–39 in Listing 2), and GAE supports none of these properties (lines 25–26 in Listing 2). This metadata allows the `PolicyEngine` to reason about these platforms.

Listing 2 First part of the deployment descriptor, specifying the different PaaS platforms in the multi-cloud and their respective properties.

```
1  <?xml version=''1.0'' encoding=''UTF-8''?>
2  <multicloud>
3    <cloudenv id=''local''>
4      <name>JBoss AS 7</name>
5      <hosted>private</hosted>
6      <location>Belgium</location>
7      <provider>
8        <company>CloudPost</company>
9        <hq>Belgium</hq>
10     </provider>
11     <properties>
12       <secureComm>true</secureComm>
13       <encrypted>false</encrypted>
14     </properties>
15   </cloudenv>
16   <cloudenv id=''GAE''>
17     <name>Google App Engine</name>
18     <hosted>public</hosted>
19     <location>US</location>
20     <provider>
21       <company>Google</company>
22       <hq>US</hq>
23     </provider>
24     <properties>
25       <secureComm>false</secureComm>
26       <encrypted>false</encrypted>
27     </properties>
28   </cloudenv>
29   <cloudenv id=''OpenShift''>
30     <name>Red Hat OpenShift</name>
31     <hosted>public</hosted>
32     <location>EU</location>
33     <provider>
34       <company>Red Hat</company>
35       <hq>US</hq>
36     </provider>
37     <properties>
38       <secureComm>true</secureComm>
39       <encrypted>true</encrypted>
40     </properties>
41   </cloudenv>
42 </multicloud>
43 <components>
44   ...
45 </components>
```

4.2 Scenario #1: Enforcing tenant-specific constraints

Potential tenants of the CloudPost system are *supermarkets* that send targeted advertisements or leaflets to their customers, *utility companies* that send out personal invoices, and *hospitals* that want to deliver the

medical reports to the doctors or patients. Although these tenants have roughly the same functional requirements (i.e. generating and delivering digitalized documents), they have different non-functional requirements, for example regarding security.

Assuming that the document generation service is deployed on each PaaS platform of the multi-cloud environment (see Listing 3), then a tenant can constrain (via policies) which instance of this application component will be used for processing his requests. Such a tenant policy specifies the required properties of a certain application component. For example, the policy in Figure 4 specifies that, for a confidential document type, the document generation service must provide encrypted storage and secure communicaton, or must run in a private cloud. Tenants can further define other document types (that are mapped to message types in the middleware) and specify constraints for each type.

Listing 3 Second part of the deployment descriptor, specifying one private and two public versions of an application component.

```
<?xml version=''1.0'' encoding=''UTF-8''?>
<multicloud>
  ...
</multicloud>
<components>
 <component id=''Doc1''>
   <interface>cloudpost.service.DocumentService</
       interface>
   <description>
     Generation and storage of documents in
         private cloud
   </description>
   <implementation>
     cloudpost.document.generation.
         DocumentServiceImpl
   </implementation>
   <cloud>local</cloud>
 </component>
 <component id=''Doc2''>
   <interface>cloudpost.service.DocumentService</
       interface>
   <description>
     Generation and storage of documents on
         Google AppEngine
   </description>
   <implementation>remote</implementation>
   <url>
     http://cloudpost-gae.appspot.com/remote/
         docservice
   </url>
   <cloud>GAE</cloud>
 </component>
 <component id=''Doc3''>
   <interface>cloudpost.service.DocumentService</
       interface>
   <description>
     Generation and storage of documents on
         OpenShift
   </description>
   <implementation>remote</implementation>
   <url>
     http://cloudpost.openshift.com/docservice
   </url>
   <cloud>OpenShift</cloud>
 </component>
</components>
```

Listing 4 Example of a tenant policy for confidential documents. Confidential documents should be processed in private clouds, or on an external location that supports encrypted storage and ensures secure communication.

```
1 Component =
2   cloudpost.service.DocumentService,
3 MessageType = confidential,
4   Location = *,
5   Access = Private
6    or
7   (Encrypted = true and SecureComm = true)
```

In case of the log management and medical image processing services, the location where data is stored and the PaaS provider are important properties (e.g. lines 6–10 in Listing 2). For example, some banks require that a private cloud is used in the same country as where the bank is located. This can easily be enforced using tenant policies that specify constraints on location and by assigning the appropriate message types to the requests and/or data. This way, tenants keep control over their data in a fine-grained way, while the SaaS provider can still benefit from the flexibility and scalability of a multi-cloud environment.

4.3 Scenario #2: Dynamically controlling spill-over

The CloudPost provider aims to maximally utilize his/her own private data center. Therefore, the provider specifies a load threshold using a spill-over policy. As long as the load in the private cloud is lower than this threshold, all incoming requests are processed by the local instance of the document generation service (i.e. component "Doc1" in Listing 3).

However, the processing of documents is often of a recurring nature, e.g. processing payslips and invoices at the end of the month, typically in the form of large document batches. The private data center of CloudPost thus faces high peaks in loads at the end of the month. To address these peak loads, the public PaaS platforms are used as spill-over. Evidently, also the decision which documents to process in the public cloud, depends on the applicable tenant policies, for example non-confidential documents (like advertising) will be generated in the public cloud, and confidential documents in public clouds that offer encrypted storage or in the private cloud.

4.4 Scenario #3: Migrating to other PaaS providers

The CloudPost provider does not want to be completely dependent on the current three PaaS providers. Therefore, he regularly evaluates new PaaS offerings, for example with respect to cost, security properties, availability and performance guarantees, etc. When one of the current PaaS providers in the multi-cloud becomes too expensive or a better alternative is available, then the PaaSHopper

middleware enables the CloudPost provider to easily replace the PaaS platforms that are part of the multi-cloud. Similarly, the multi-cloud can be extended with additional PaaS offerings, for example when a tenant has a feature request that is unsupported by the current platforms.

More specifically, the SaaS provider should only update the part of the deployment descriptor that defines the multi-cloud environment (cf. Listing 2), and the PaaSHopper middleware will automatically adapt to the new environment. As the measurements of the migration overhead in [19] show, the abstraction layer enables the migration of the CloudPost application to different PaaS platforms without any impact on the application code. All code changes are contained within the portability drivers. This means that to support a new PaaS platform in the multi-cloud, merely the appropriate portability drivers have to be installed. Furthermore, no policy changes are required, as the policies only use properties to specify constraints and do not refer to specific PaaS platforms. Finally, this adaptation occurs instantaneous as the policy evaluation is applied at the fine-grained level of requests.

4.5 Concluding remarks

The validation shows (i) that the abstraction layer of the PaaSHopper middleware supports the migration of the document processing application across multiple PaaS platforms, without any code modification (at the application level), and (ii) the effectiveness of using policies to control the execution and storage in a fine-grained way and in correspondence to the different stakeholder-specific requirements.

Although the validation focuses on a single, representative application (i.e. document processing), we did analyse the feasibility and applicability of the middleware for the four application cases presented in Section 2.1. Moreover, the complexity of this work lies in the middleware and the validation demonstrates that it supports the deployment of the document processing application on top of quite different PaaS offerings (i.e. JBoss AS, GAE [5] and OpenShift [27]).

However, further validation and improvement of the PaaSHopper middleware is required to verify its effectiveness and extensibility in the context of different SaaS applications as well as different PaaS platforms and their provided services (especially storage). For example, we did not cover interactive applications, but we believe that the proposed approach is still viable, with the dispatcher acting as a policy-driven load balancer for all incoming requests. Similarly, the expressiveness of the policies should be further evaluated.

Section 3 presented the open and versatile architecture of the PaaSHopper middleware. The document processing prototype validates only one deployment instance of this architecture, with a single dispatcher deployed in the private cloud. We believe that this is the most common deployment of the PaaSHopper middleware, but it is certainly worth to investigate the potential benefits and/or issues of having multiple dispatchers and policy engines within a multi-cloud environment.

5 Discussion

This section discusses the strengths and limitations of the current PaaSHopper middleware. Based on this discussion and our experience with this middleware in the different SaaS application cases, we itemize a set of open challenges and directions for future work.

5.1 Reconfigurable policy evaluation

The PaaSHopper middleware is able to control the processing and/or storage of data in a fine-grained way by enforcing policies on any interaction between the different application components. However, as indicated by the application cases in Section 2, there exist different types of SaaS applications, such as data-driven, computationally-intensive or combinations. Furthermore, the different stakeholders can specify policies with respect to the location of processing, the location of storage, the load etc. Depending on the application type, it is not always desirable to intercept each interaction for policy evaluation. This not only leads to a performance overhead because of the additional (possibly remote) policy evaluation, but in data-driven applications such as medical image processsing, this can lead to sending back and forth large data sets between different cloud offerings.

In the current prototype, this problem is solved by logically grouping different application components as one entry in the deployment descriptor. For example, we specified that the document generation service includes the generation as well as the storage of the generated documents. This ensures that the policies apply to both processing and storage, and thus prevents that input data is sent to a public PaaS platform for generation and then sent back to the private cloud for storage. However, this requires the SaaS provider to adapt the granularity of the components or services, and it limits the flexibility.

Alternatively, the PaaSHopper middleware could be extended to enable the reconfiguration of the interception points for policy evaluation. This requires an application model that defines the dependencies between the different application components. When set to limit the amount of interactions across the multi-cloud environment (e.g. data-driven application), the policy engine can take all dependencies into account for incoming requests and make a single decision on where the entire execution will take place. Only when necessary, the policy evaluation occurs at every interaction, for example

for the simulation processing application, where the data traffic is limited and processing may occur in the public cloud.

5.2 Dynamic deployment specification

The PaaSHopper middleware supports different types of policies based on a potentially large variety of properties. In addition, the SaaS provider can further extend (i) the specification of the PaaS platforms in the deployment descriptor with domain- and application-specific properties, and (ii) the middleware with custom policy engines to support new types of policies. In the current prototype, these properties are considered to be static.

However, we can think of several interesting multi-cloud scenarios that involve dynamic cloud properties, such as performance, availability and cost. These properties can change over time and also vary depending on the type of operation. The use of dynamic properties also requires more expressive policies to take into account the cost and performance of the different PaaS offerings.

To support these dynamic cloud properties, the deployment specification has to be updated at run time. This requires, as depicted in Figure 5, the integration of the PaaSHopper middleware with cloud management services to form a control loop. This enables the middleware (i) to continuously monitor the application as well as the underpinning PaaS environment(s), (ii) to use this monitoring data to improve decision making, and (iii) to dynamically reconfigure and (re)deploy the SaaS application across the multi-cloud environment (e.g. in case of a spill-over or migration to another PaaS offering) if necessary, instead of a static assignment as is currently the case in our prototype. The configuration management service then has to send the up-to-date deployment view to the dispatcher. With such a control loop, the PaaSHopper middleware actually becomes a cloud broker.

5.3 Retrieving data in multi-cloud

Our experience with the deployment of the different application cases in a multi-cloud environment revealed a major challenge with respect to locating and querying data. When an application wants to access data that is previously stored, it has to know where it is stored in the multi-cloud environment. Moreover, to prevent unnecessary data traffic, access to the data should preferably occur on the same cloud platform. This problem arises especially in dynamic multi-cloud environments, for example with spill-over scenarios: during the spill-over data is stored using a public cloud storage service, but afterwards all execution should occur locally in the private cloud.

Depending on the application type and the amount of data, different solutions are possible. For example, in the case of the document processing application, an application-level index of all documents can be

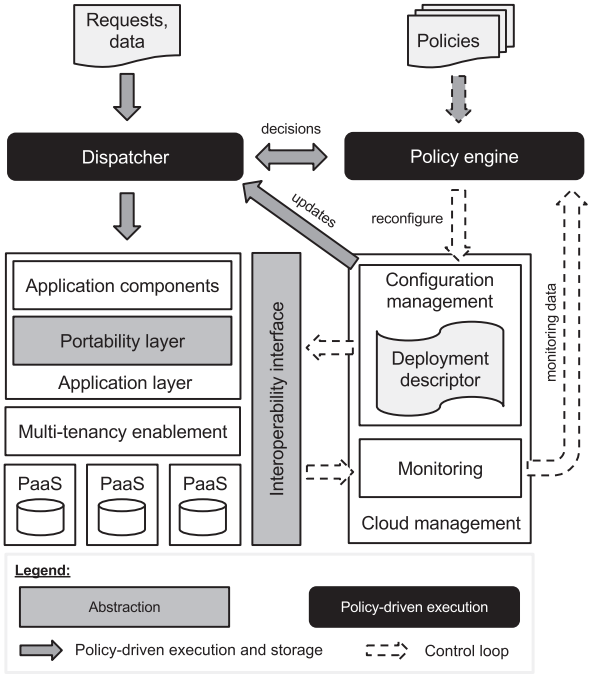

Figure 5 PaaSHopper integrated with cloud management. High-level overview of the policy-driven PaaSHopper middleware, including the integration with cloud management services such as monitoring and configuration management.

maintained. Furthermore, data can regularly be migrated to aggregate the data of the same tenant. However, an index is not appropriate for the log management service, as this application involves a continuous input stream of small log entries and the analysis of all log entries (i.e. the full data set). As this application case uses a set of private data centers, the policy engine of the PaaSHopper middleware could be used to ensure that the logs of the same tenant are stored at the same location, and that the analysis is performed where the data is stored.

5.4 Portability and interoperability

In previous work [7], we identified three categories of PaaS platforms with respect to SaaS development. The first and second category contain the PaaS platforms that support traditional programming models and aim to match the APIs of enterprise application servers and middleware platforms (e.g. CloudFoundry [28], OpenShift [27] and GAE [5]). However, these platforms still use vendor-specific solutions, especially for typical cloud services such as scalable storage, which hinders portability. The third category consists of metadata-driven PaaS platforms. These platforms use a higher-level composition and configuration interface, and lack any compatibility with common programming models.

The PaaSHopper middleware is aimed at (Java-based) PaaS platforms of the first and second category, and offers an abstraction layer to address the heterogeneity and to support migration. Evidently, the appropriate portability drivers should be present. Indirectly, the middleware also supports IaaS offerings through the use of application servers.

As exemplified by the different application cases, the PaaSHopper middleware does not put any constraints on the types of applications that are supported. However, to enable fine-grained control of the execution, the application should be decomposable into *modular and loosely-coupled* software artifacts (e.g. components, services). This way, the PaaSHopper middleware can intercept the incoming requests as well as the invocations between the different components or services within the application, and dynamically adapt the composition driven by the policies that are applicable. Such an application model is commonly supported, for example by component-based software development (CBSD) [29], aspect-oriented software development (AOSD) [30], dependency injection [25], and service-oriented computing (SOC) [31,32].

Finally, interoperability is not only a concern with respect to the interaction between the different application components across the heterogeneous multi-PaaS environment. When integrating with cloud management services such as monitoring and configuration management, interoperability of these services with the different PaaS platforms is also required. With the large variety of management APIs of these platforms, there is a need for a uniform interface (and accompanying implementations) to address the heterogeneity and to support the management of multi-cloud environments, as indicated by Figure 5. This is certainly an open challenge.

5.5 Versatility of the PaaSHopper middleware

This paper presents a generic middleware framework in the sense that it provides a versatile architecture that supports different implementations and deployments, depending on the application type and the specific nature of the multi-cloud environment.

One of these supported customizations is related to the deployment of the dispatcher. For example, the document processing application is deployed in a dynamic multi-cloud environment: the public cloud environments are only used when the load on the on-premise infrastructure is high. Therefore, the entry point for the application is deployed in the private cloud only. Consequently, we decided to use one dispatcher, which is also deployed in the private cloud. A similar deployment is recommended for the simulation processing case (cf. application #4).

However, in the case of the log management service or medical image processing (cf. applications #2 and #3), the different cloud environments are always used (i.e. a static multi-cloud environment), so it is possible to have multiple entry points and then it might be recommended to have a dispatcher in every data center to limit the latency overhead. In future work, we will investigate the impact of multiple dispatchers within a multi-cloud environment on the operation of the PaaSHopper middleware as well as the application.

A second customization lies in the optional usage of the abstraction layer. The proposed abstraction layer can be interchanged with other solutions for cloud portability (cf. Section 3.1) or can even be omitted. The latter is especially relevant when the PaaS platforms in the multi-cloud environment offer the same or technologically compatible APIs.

6 Related work

This section discusses three domains of related work: a) multi-cloud support, b) policy-driven middleware, and c) cloud brokerage.

6.1 Multi-cloud support

Mietzner et al. [33] focus on cloud application portability, using an extension to the service component architecture (SCA) with variability descriptors and multi-tenancy patterns. However, SCA applications can only be executed in an SCA application environment, whereas our work focuses on PaaS platforms, which typically do not support SCA. Moreover, their focus is on multi-tenant

customization, packaging, and deployment migration obstacles, while this work presents a policy-driven middleware that offers a common PaaS API in order to facilitate flexible, reconfigurable deployment and execution of SaaS applications in multi-PaaS environments.

Paraiso et al. [11] present a so-called federated multi-cloud PaaS infrastructure that enables the deployment of service component architecture (SCA) applications on heterogeneous PaaS and IaaS offerings. This federated PaaS infrastructure relies on their own FraSCAti execution environment for SCA applications. Furthermore, it supports the dynamic reconfiguration of component bindings and the addition of components and services, but this happens globally for all tenants. In contrast, we focus on a technology-agnostic approach without the need of a SCA execution environment, and the PaaSHopper middleware supports dynamic reconfiguration on a per-tenant basis and is driven by co-existing policies that express the stakeholder-specific constraints.

Cunha et al. [34] proposed a middleware architecture that facilitates dynamic deployment, registration and portability of services across different PaaS providers, and enables developers to create and expose services using a cloud-based service delivery platform within a service-oriented architecture (SOA). Although, the authors also focus on portability and application migration, there are still significant differences with our work: (i) their focus is on SOA applications, while the PaaSHopper middleware does not put any constraints on the application model, (ii) the authors acknowledge the need for a uniform PaaS API, but do not make any concrete suggestions whereas we have implemented a common abstraction layer for three heterogeneous PaaS platforms, and (iii) their solution does not enable the different stakeholders to control the execution of the deployed applications.

Another related solution is provided by the European mOSAIC project [35], where an independent PaaS platform API is developed to provide support for heterogeneous hybrid clouds. The developed API uses a driver architecture and can be deployed on top of heterogeneous hybrid PaaS platforms. We applied a similar approach for the abstraction layer, but with different focus. While they have focused on a PaaS API for task automation, the PaaSHopper middleware offers a uniform API for three common cloud services, including structured storage (NoSQL), blob storage and asynchronous task execution. Moreover, the focus of our work is on providing the different stakeholders more control over the execution of multi-tenant SaaS applications in multi-PaaS environments.

DRACO [36] is a new PaaS platform that is inspired by FCAPS, the ISO telecommunications management network model. It is built on top of an IaaS layer and can be utilized by other SaaS applications or PaaS platforms. The focus of DRACO is (i) to address issues concerning PaaS management such as fault tolerance, configuration, accounting, performance, and security, and (ii) to provide a platform for the development of algorithms that require parallel processing and a considerable amount of computation in the cloud. In contrast, we solve application-level issues and provide portability and interoperability support across existing PaaS environments in order to support fine-grained control of execution and storage.

Kaviani et al. [37] have proposed a cross-tier partitioning approach to support developers making the trade-off between performance and cost in hybrid clouds. The focus is on optimizing the partitioning of both the application- and data-tier of web applications across hybrid IaaS deployments driven by application profiles, while taking into account that sensitive data may not be moved to the public cloud. In contrast, the PaaSHopper middleware does not rely on profiling to make decisions, but enforces at run time the different tenant- and provider-specific policies on the different interactions. Furthermore, we assume that the SaaS provider has already invested in a private cloud and wants to maximally utilize this on-premise infrastructure. However, the work by Kaviani et al. can be used to extend the PaaSHopper middleware, for example to limit the costs and performance overhead because of data traffic (as discussed in Section 5).

Petcu et al. [22] have conducted a survey on the state of the art with respect to portability of applications that consume cloud services. The paper provides a taxonomy for cloud portability, an overview of the latest solutions, and it identifies the open challenges. The use of abstraction layers and adapters to hide the differences and to expose a uniform API, cf. our approach, is one of the common solutions they identified to address the portability issue. Kolb et al. [38] have defined a model that describes the current PaaS offerings, and they clustered a set of core properties into a PaaS profile in order to support the comparison and portability matching of PaaS offerings based on application dependencies and capabilities.

6.2 Policy-driven middleware

Policy-driven middleware is commonly used in the context of service compositions to support customizaton and dynamic selection of web services, for example [39,40]. These policy-driven adaptations are required to keep fulfilling the QoS requirements of the applications. In the PaaSHopper middleware, however, policies are used to constrain the deployment and execution of multi-cloud applications based on the properties of the underpinning platforms. Furthermore, the focus is on keeping control over the applications and the data, and less on QoS.

In [41], the authors design a middleware that supports the development of multi-tenant SOA applications. In such applications both tenants and application providers can define policies. The middleware uses a message dispatch mechanism in order to guide incoming request messages to the right service instance as indicated by the global and tenant-specific policies. The relevant policies are processed in two steps. In the first step, all the global policies are applied, and in the second step tenant-specific policies are taken into account. This paper, however, focuses on the use of policies in a multi-cloud context, thus containing multiple run-time environments. These policies are then used to select a PaaS platform within the multi-cloud. Furthermore, the PaaSHopper middleware supports more complex policies that relate to the active tenant as well as the current message type.

6.3 Cloud brokerage

Several European research projects tackle challenges with respect to cloud brokerage, for example OPTIMIS [42] and MODAClouds [43]. The focus of OPTIMIS is on optimal placement of virtual machines (VMs) in multi-cloud environments driven by cost, energy efficiency, QoS, etc., and also involves SLA negotiation and creation. MODAClouds aims to support cross-cloud portability and to automatically (re)configure the deployment of applications on multi-clouds to ensure the QoS. The goal of the latter project is certainly related to this work. Although the PaaSHopper middleware is not a cloud broker, it does provide several complementary solutions to the MODAClouds project, for example an abstraction layer to support the cross-cloud migration of multi-tenant SaaS applications, and a policy-driven middleware layer to control the processing and storage of data in a fine-grained way.

7 Conclusion

Multi-cloud deployment has the potential of solving many problems that enterprises currently face with cloud computing. Especially in the area of multi-tenant SaaS applications, there is a large potential in leveraging not one, but many different underpinning PaaS platforms, each potentially having different properties in terms of cost, performance, availability, security, etc.

To realize this potential however, complex middleware support is required to deal with issues of portability and interoperability and to provide the different stakeholders (tenant, SaaS provider, etc.) with fine-grained control on the deployment, execution and operation of the SaaS offering on top of multiple PaaS platforms.

This paper presents such a middleware architecture for exploiting SaaS applications in a multi-PaaS environment. This middleware offers on the one hand a PaaS abstraction layer that ensures portability accross different PaaS platforms, while on the other hand it offers policy-based control mechanisms to influence its execution. These policies are based on expressive abstractions and as such allow defining key non-functional requirements such as constraints about the geographical deployment location, security constraints, etc.

Although the presented middleware is a first step in the development of middleware systems that deal with the complexity and heterogeneity inherent to hybrid and multi-cloud environments, many research opportunities are left open. In future work, we plan to investigate additional adaptation support to dynamically (un)deploy components on the appropriate public platforms and enabling more expressive policies based on more dynamic cloud properties such as cost, performance, etc.

Competing interests

The authors declare that they have no competing interests.

Authors' contributions

SW has been in charge of defining the PaaSHopper middleware and drafting the manuscript. Further refinement was conducted in collaboration with DVL. The validation was conducted by AR. BL and WJ have participated in the initial design of the middleware, and have contributed significantly to the positioning of the work. All authors have read and approved the final manuscript.

Acknowledgements

We would like to thank Tom Desair for the initial implementation of the PaaSHopper middleware as part of his master thesis. This research is partially funded by the Research Fund KU Leuven (project GOA/14/003 - ADDIS), the FWO project iSPEC and by the iMinds DMS2 project, which is co-funded by iMinds (Interdisciplinary institute for Technology), a research institute founded by the Flemish Government. Companies and organizations involved in the project are Agfa Healthcare, Luciad, UP-nxt and Verizon Terremark, with project support of IWT (government agency for Innovation by Science and Technology).

Received: 16 September 2014 Accepted: 25 November 2014
Published online: 29 January 2015

References

1. Mell P, Grance T (2011) The nist definition of cloud computing. Special publication 800-145, National Institute of Standards and Technology (NIST). http://csrc.nist.gov/publications/nistpubs/800-145/SP800-145.pdf
2. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, Lee G, Patterson D, Rabkin A, Stoica I, Zaharia M (2010) A view of cloud computing. Commun ACM 53(4):50–58. doi:10.1145/1721654.1721672
3. Zhang Q, Cheng L, Boutaba R (2010) Cloud computing: State-of-the-art and research challenges. J Internet Serv Appl 1(1):7–18. doi:10.1007/s13174-010-0007-6
4. Amazon Web Services LLC Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/. [Last visited on November 24, 2014]
5. Google Inc. Google App Engine. https://cloud.google.com/appengine/docs. [Last visited on November 24, 2014]
6. Salesforce.com Inc. Salesforce CRM. http://www.salesforce.com/. [Last visited on November 24, 2014]
7. Walraven S, Truyen E, Joosen W (2014) Comparing PaaS offerings in light of SaaS development. Computing 96(8):669–724. doi:10.1007/s00607-013-0346-9
8. Mueller D (2014) It's 2014 and PaaS is eating the world. https://www.openshift.com/blogs/its-2014-and-paas-is-eating-the-world
9. Leavitt N (2009) Is cloud computing really ready for prime time? Computer 42(1):15–20. doi:10.1109/MC.2009.20

10. Marston S, Li Z, Bandyopadhyay S, Zhang J, Ghalsasi A (2011) Cloud computing — the business perspective. Decis Support Syst 51(1):176–189. doi:10.1016/j.dss.2010.12.006
11. Paraiso F, Haderer N, Merle P, Rouvoy R, Seinturier L (2012) A federated multi-cloud PaaS infrastructure. In: CLOUD '12: IEEE 5th International Conference on Cloud Computing. IEEE. pp 392–399. doi:10.1109/CLOUD.2012.79
12. Whittaker Z (2012) Amazon cloud down; reddit, github, other major sites affected. ZDNet.com. http://www.zdnet.com/article/amazon-cloud-down-reddit-github-other-major-sites-affected/. [Last visited on November 24, 2014]
13. Takahashi D (2011) Amazon's outage in third day: debate over cloud computing's future begins. VentureBeat. http://venturebeat.com/2011/04/23/amazonsoutage-in-third-day-debate-over-cloud-computings-future-begins/. [Last visited on November 24, 2014]
14. Bilton N (2012) Amazon web services knocked offline by storms. The New York Times (Bits). http://bits.blogs.nytimes.com/2012/06/30/amazon-web-services-knocked-offline-by-storms/. [Last visited on November 24, 2014]
15. Chong F, Carraro G (2006) Architecture strategies for catching the long tail. Microsoft Corporation. http://msdn.microsoft.com/en-us/library/aa479069.aspx
16. Guo CJ, Sun W, Huang Y, Wang ZH, Gao B (2007) A framework for native multi-tenancy application development and management. In: CEC/EEE '07: 9th IEEE International Conference on E-Commerce Technology and 4th IEEE International Conference on Enterprise Computing, E-Commerce, and E-Services. IEEE. pp 551–558. doi:10.1109/CEC-EEE.2007.4
17. Leavitt N (2013) Hybrid clouds move to the forefront. Computer 46(5):15–18. doi:10.1109/MC.2013.168
18. Desair T, Joosen W, Lagaisse B, Rafique A, Walraven S (2013) Policy-driven middleware for heterogeneous, hybrid cloud platforms. In: ARM '13: Proceedings of the 12th International Workshop on Adaptive and Reflective Middleware. ACM, New York, NY, USA. pp 7–12. doi:10.1145/2541583.2541585
19. Rafique A, Walraven S, Lagaisse B, Desair T, Joosen W (2014) Towards portability and interoperability support in middleware for hybrid clouds. In: CrossCloud '14: Proceedings of the 1st IEEE INFOCOM CrossCloud Workshop. IEEE. pp 7–12. doi:10.1109/INFCOMW.2014.6849160
20. Decat M, Bogaerts J, Lagaisse B, Joosen W (2014) The e-document case study: Functional analysis and access control requirements. CW Reports 654. https://lirias.kuleuven.be/handle/123456789/440202
21. Petcu D (2011) Portability and interoperability between clouds: Challenges and case study. In: ServiceWave '11: Towards a Service-Based Internet. Springer, Berlin, Heidelberg. pp 62–74. doi:10.1007/978-3-642-24755-2_6
22. Petcu D, Vasilakos AV (2014) Portability in clouds: Approaches and research opportunities. Scalable Comput Pract Exp 15(3):251–270. doi:10.12694/scpe.v15i3.1019
23. Red Hat Inc. Hibernate Object/Grid Mapper (OGM). http://hibernate.org/ogm/. [Last visited on November 24, 2014]
24. Impetus Technologies Inc. Kundera: Object-datastore mapping library for NoSQL datastores. https://github.com/impetus-opensource/Kundera. [Last visited on November 24, 2014]
25. Fowler M (2004) Inversion of control containers and the dependency injection pattern. http://martinfowler.com/articles/injection.html
26. Walraven S, Truyen E, Joosen W (2011) A middleware layer for flexible and cost-efficient multi-tenant applications. In: Middleware '11: Proceedings of the 12th ACM/IFIP/USENIX International Conference on Middleware. Springer, Berlin, Heidelberg. pp 370–389. doi:10.1007/978-3-642-25821-3_19
27. Red Hat Inc. Red Hat OpenShift. https://www.openshift.com/. [Last visited on November 24, 2014]
28. VMware Inc. Cloud Foundry. http://www.cloudfoundry.org/. [Last visited on November 24, 2014]
29. Szyperski C (2002) Component software - beyond object-oriented programming. 2nd edn. Addison-Wesley/ACM Press, Boston, MA, USA
30. Filman RE, Elrad T, Clarke S, Akşit M (2004) Aspect-oriented Software Development. 1st edn. Addison-Wesley Professional, Boston, MA, USA
31. Papazoglou MP (2003) Service-oriented computing: Concepts, characteristics and directions. In: WISE '03: Proceedings of the Fourth International Conference on Web Information Systems Engineering. IEEE. pp 3–12. doi:10.1109/WISE.2003.1254461
32. Huhns MN, Singh MP (2005) Service-oriented computing: Key concepts and principles. IEEE Internet Computing 9(1):75–81. doi:10.1109/MIC.2005.21
33. Mietzner R, Leymann F, Papazoglou MP (2008) Defining composite configurable SaaS application packages using SCA, variability descriptors and multi-tenancy patterns. In: ICIW '08: 3rd International Conference on Internet and Web Applications and Services. IEEE. pp 156–161. doi:10.1109/ICIW.2008.68
34. Cunha D, Neves P, Sousa PNMd (2012) Interoperability and portability of cloud service enablers in a PaaS environment. In: CLOSER '12: Proceedings of the 2nd International Conference on Cloud Computing and Services Science. SciTePress. pp 432–437. doi:10.5220/0003959204320437
35. Petcu D, Macariu G, Panica S, Crăciun C (2013) Portable cloud applications - from theory to practice. Future Generation Comput Syst 29(6):1417–1430. doi:10.1016/j.future.2012.01.009
36. Celesti A, Peditto N, Verboso F, Villari M, Puliafito A (2013) DRACO PaaS: A distributed resilient adaptable cloud oriented platform. In: IPDPSW '13: IEEE 27th International Parallel and Distributed Processing Symposium Workshops PhD Forum. IEEE. pp 1490–1497. doi:10.1109/IPDPSW.2013.266
37. Kaviani N, Wohlstadter E, Lea R (2013) Cross-tier application and data partitioning of web applications for hybrid cloud deployment. In: Middleware '13: Proceedings of the ACM/IFIP/USENIX 14th International Middleware Conference. Springer, Berlin, Heidelberg. pp 226–246. doi:10.1007/978-3-642-45065-5_12
38. Kolb S, Wirtz G (2014) Towards application portability in Platform as a Service. In: SOSE '14: Proceedings of the 8th IEEE International Symposium on Service-Oriented System Engineering. IEEE. pp 218–229. doi:10.1109/SOSE.2014.26
39. Wohlstadter E, Tai S, Mikalsen T, Rouvellou I, Devanbu P (2004) GlueQoS: Middleware to sweeten quality-of-service policy interactions. In: ICSE '04: Proceedings of the 26th International Conference on Software Engineering. IEEE Computer Society, Washington, DC, USA. pp 189–199. doi:10.1109/ICSE.2004.1317441
40. Erradi A, Maheshwari P, Tosic V (2006) Policy-driven middleware for self-adaptation of web services compositions. In: Middleware '06: ACM/IFIP/USENIX 7th International Middleware Conference. Springer, Berling, Heidelberg. pp 62–80. doi:10.1007/11925071_4
41. Azeez A, Perera S, Gamage D, Linton R, Siriwardana P, Leelaratne D, Weerawarana S, Fremantle P (2010) Multi-tenant SOA middleware for cloud computing. In: CLOUD '10: IEEE International Conference on Cloud Computing. IEEE Computer Society, Washington, DC, USA. pp 458–465. doi:10.1109/CLOUD.2010.50
42. Ferrer AJ, Hernández F, Tordsson J, Elmroth E, Ali-Eldin A, Zsigri C, Sirvent R, Guitart J, Badia RM, Djemame K, Ziegler W, Dimitrakos T, Nair SK, Kousiouris G, Konstanteli K, Varvarigou T, Hudzia B, Kipp A, Wesner S, Corrales M, Forgó N, Sharif T, Sheridan C (2012) OPTIMIS: A holistic approach to cloud service provisioning. Future Generation Comput Syst 28(1):66–77. doi:10.1016/j.future.2011.05.022
43. Ardagna D, Di Nitto E, Mohagheghi P, Mosser S, Ballagny C, D'Andria F, Casale G, Matthews P, Nechifor CS, Petcu D, Gericke A, Sheridan C (2012) MODAClouds: A model-driven approach for the design and execution of applications on multiple clouds. In: MiSE '12: ICSE Workshop on Modeling in Software Engineering. IEEE. pp 50–56. doi:10.1109/MISE.2012.6226014