J Netw Syst Manage
DOI 10.1007/s10922-012-9260-2

The Building Blocks of a PaaS

**Cláudio Teixeira · Joaquim Sousa Pinto ·
Ricardo Azevedo · Tiago Batista · André Monteiro**

Received: 15 June 2011 / Revised: 8 December 2012 / Accepted: 26 December 2012
© The Author(s) 2013. This article is published with open access at Springerlink.com

Abstract Traditional cloud computing providers enable developers to program and deploy applications in the cloud by using platform as a service (PaaS) providers. Despite the benefits of such an approach, this service model usually comes with a high level of the lock in effect into the service provider. The lock in effect occurs when a software developer needs to change its cloud provider. When this happens, it usually means a major application rewrite, specific for the new PaaS. This paper details the initial results of a project whose goal is to build a PaaS where vendor lock in does not occur. With this PaaS, developers wishing to deploy their applications into the cloud may continue to program using their usual development environments. There are no necessary changes required to make the application PaaS compatible. As a proof of concept, we developed an open source PaaS management application as a regular web application and then deployed it on the cloud.

This is an original, unpublished work. Neither the entire paper nor any part of its contents has been published or has been accepted for publication elsewhere. This work has not been submitted to any other journal.

C. Teixeira (✉) · J. S. Pinto · T. Batista · A. Monteiro
DETI, Universidade de Aveiro, Campus Universitário de Santiago, 3810-193 Aveiro, Portugal
e-mail: claudio@ua.pt

J. S. Pinto
e-mail: jsp@ua.pt

T. Batista
e-mail: tsbatista@ua.pt

A. Monteiro
e-mail: andremonteiro@ua.pt

R. Azevedo
PT Inovação, Rua Eng. José Ferreira Pinto Basto, 3810-106 Aveiro, Portugal
e-mail: ricardo-a-pereira@ptinovacao.pt

Published online: 19 January 2013

Keywords Platform as a service · Cloud computing · Performance testing

1 Introduction

Cloud computing has evolved from previous computing paradigms using concepts that go back to the days of mainframes [1]. Its rapid and scattered development originated a huge usage momentum; however, its accelerated growth lacked a sustaining basis. With this scattered development and with few or no defined standards, each provider devised its own cloud architecture and infrastructure, a very similar situation to the appearance of the TCP [2].

Nevertheless, some efforts are being made to standardize cloud computing development [3, 4]. These standards would leverage the cloud-based environment development. As part of this long-lasting process, PaaS is still in a maturing process. As illustrated in Fig. 1, PaaS is in an early phase, especially when compared with other cloud components' development stages. As a result, there is no PaaS available that simultaneously fulfills the users' needs, budget, and system adaptability, although the expectations to get such PaaS providers are high.

As the first private cloud implementations were steady and stable, developers turn their minds to providing private PaaS. One of the main objectives of these private PaaS projects was to achieve control over the bursting of virtual machines (VMs) and their underlying virtual networks, whose management was completely out of sight on commercial solutions.

Fig. 1 Hype cycle for cloud computing, 2011 [5]

The solutions pertinent to this article are those that fall under the definition of PaaS and SaaS as proposed by NIST [6]. For an in-depth review of cloud computing terms and taxonomy, please refer to [7].

When looking at the current IaaS and PaaS offerings, solutions lack interoperability. This issue is being addressed by projects such as Deltacloud [8] and Aeolus [9] on the IaaS front. While this approach eases the interoperability and interchange of data and operating systems images between clouds, it does not fully remove the burden of infrastructure management from the end user. For seamless handling of the infrastructure management problem, the cloud client needs a different service model, the PaaS. This service model allows automatic and seamless application scaling without the need of intervention from the client or the application.

PaaS implementations range from application specific to generic application programming solutions. However, up to the moment, most PaaS also have the downside of requiring a proprietary software development kit (SDK) that will cause the resulting application to be tightly locked to the supplier's runtime or development environment [10].

The work presented in the paper follows the authors' previous work [11], where a distributed cloud computing scenario was proposed and PaaS was deemed as the ideal abstraction layer from the point of view of a software developer. The previous work identifies the key stakeholders on the distributed architecture, and proposes possible streams of revenue or rewards for each player [12]. An initial architectural model is also proposed and extensively described by Batista [13].

We propose to build a new PaaS, capable of supporting several cloud implementations that do not lock developers into a specific SDK. The expected result is a fully transparent PaaS runtime that requires no special SDK. This would enable any Java written application to deploy on the cloud and benefit from the cloud's features, without having to specifically write any code or use any SDK for that matter.

This paper presents architectural details and implementation consideration of some of the PaaS components already developed. With the implemented components, we present some preliminary functionality and performance testing.

The organization of this paper is as follows: Sect. 1 introduces the problem that we are currently tackling and Sect. 2 presents the state of the art relevant to the problem. Section 3 presents and discusses the proposed PaaS architecture, while Sect. 4 provides details on the implementation of the proposed PaaS. Section 5 presents the testbed and results of the PaaS usage and Sect. 6 presents the conclusions on this PaaS architecture.

2 State of the Art

2.1 Cloud Computing Layers

Usually, cloud computing is divided in three distinct logic layers that are dependent of their main function. These are infrastructure, platform and service. Any of these layers can be publicly available to the end-user, according to its needs. Figure 2

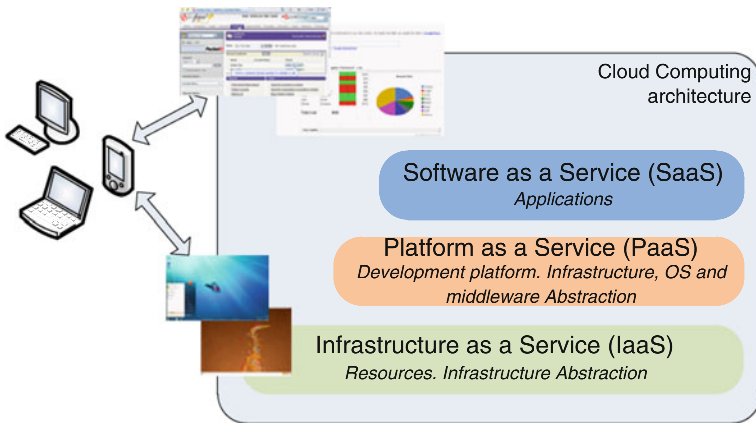

Fig. 2 Cloud computing architecture

graphically presents the three-layer vision on the cloud computing architecture. Next, sub-sections briefly describe each layer, with a special focus on the infrastructure as a service stratum that PaaS, the main motivation of this paper, relies on.

2.1.1 Software as a Service (SaaS)

The software layer allows the access to platform-enabled applications by typically using a web browser. Software as a service (SaaS) is usually available on cloud solutions.

The SaaS model is not new [14]; the concept began to gain some visibility in the late 90s. With the rise of cloud computing, this model was finally able to reach mainstream. Currently, even the most traditional software houses are developing products that embrace this business model [15, 16].

2.1.2 Platform as a Service (PaaS)

The platform layer consists of specially crafted software development kits (SDK) that allow developers to produce their own programs using all of the cloud-supported features. In a common scenario, the SDK distributed by the service provider is aware of the cloud capabilities but abstracts them from the user. When deployed on the platform, an application automatically takes advantage of the scaling features available.

The downside to this approach is vendor lock in. There is no standard cloud SDK or programming language. This means that from the moment a developer chooses to use features from a given SDK, the only way to switch vendors is to redesign the parts of the application that rely on the SDK functionality [17].

2.1.3 Cloud Infrastructure as a Service (IaaS)

Both PaaS and SaaS rely on IaaS. Before discussing the particularities of PaaS, we must first understand the bottom layer of the cloud computing architecture: IaaS. Searches for projects that implement the required functionality showed that a great number of them started as grid management platforms and are now turning to cloud interfaces, leveraging previous knowledge on distributed computing platforms.

The infrastructure layer is the lower level layer, abstracting and usually virtualizing the hardware layer. There are different virtualization concepts targeting operating systems, hardware, CPUs and embedded systems, networks and storage. These enable a homogenous virtualized environment on which to install and execute specific software. Furthermore, this is the scale in and scale out layer. If required, extra infrastructure nodes can be deployed to overcome computing needs. When not needed, these nodes can again be decommissioned. The infrastructure can be made available as a service (IaaS—Infrastructure as a Service), where users may gain access to virtualized servers.

This service allows a user to provision and spawn an infrastructure and run his selected software stack. This is an on-demand service, as the infrastructure can grow or shrink with the load fluctuations. It is usually the client's responsibility to manage most of the software stack and make sure that the application running on the cloud can scale horizontally with the addition of new nodes. The startup and shutdown process of extra nodes is requested via an interface (usually a web service). When in a commercial business model, the charges are calculated on CPU/h usage. A storage service is usually associated with an IaaS, where charges are based on GB transferred and/or stored per month.

2.2 Current Trends: Sky Computing

Cloud computing abstracts resources and facilitates its usage seamlessly. Going one step further, the same principle applies to a group of clouds where several building blocks underpin the creation of a sky environment [18]. By blending resources on public and private clouds, the Sky Cloud can make intensive computing efficiently, mitigate bottlenecks, and keep sensitive data protected.

A decisive factor in favor of sky computing is the alleged freedom of choice for developers and customers alike. Using this paradigm, applications will be able to move freely among cloud providers, with no need for rewrites or code changes. The big challenge here is managing the combined resources of available clouds, public or private, and interoperating between them [19].

Because of deficient standardization, several projects try to solve the interoperating problem through middleware. In [20], authors go through the specifics of projects like libcloud, DeltaCloud, jclouds, fog and Altocumulus. A proposed architecture, using a middleware API to interoperate between clouds is depicted in Fig. 3.

On a sky computing architecture, users try to take the maximum advantage of the better (and less expensive) cloud providers available. In such scenarios, the ultimate goal is the development of a PaaS able to abstract the different cloud providers and

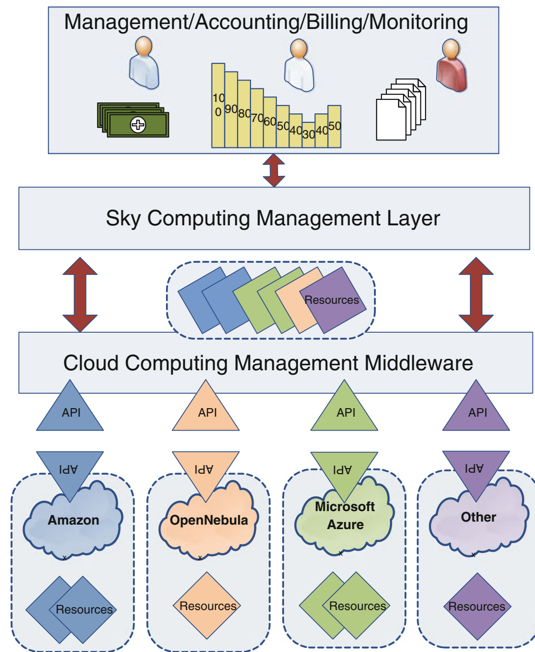

Fig. 3 Sky computing architecture [19]

give the user a single interface with all systems. Such a layer would certainly pass the frontier of its definition and would mix with some upper layer services (management, accounting, etc.).

2.3 Platform as a Service

In the last couple of years, the PaaS layer has seen a major boost in terms of solutions and efforts of standardization. For application development (where developers are familiar with middleware, frameworks, SDKs and APIs that solve most of the platform dependency issues), the PaaS model presents itself as the natural development environment.

The criteria for electing the following PaaS solutions for review are based on the solution's maturity level and functional similarities with the proposed goals for our PaaS implementation.

Despite our efforts, most of the research done is based on the providers white papers (pure commercial providers) and open-source projects (mixture of commercial providers and technology enforcers) readings.

2.3.1 Manjrasoft's Aneka

Aneka [21] is a .NET based, multi-platform PaaS implementation produced by Manjrasoft [22]. It is not a free product; therefore, no experiments were performed on this software.

This product features a very complex resource scheduling and reservation algorithm that one would expect on a grid middleware instead of on a cloud PaaS. It also features a very strong authentication mechanism used throughout its inner services. Due to its programming models, a traditional .NET application requires some porting before it can take advantage of the features of the platform. The application data is stored on a RDBMS that is connected to the platform via a plugin, meaning that alternative data storage models are a possibility [23, 24].

The provisioning model used by this project is heavily market oriented. It allows complex accounting and the establishment of service level agreements (SLA) with clients [23]. This is a major advantage if Aneka is to be used to host a public PaaS. Its usage of an RDBMS, however, may hinder such a wide public deployment.

Although Aneka has some market-oriented features, so far no one created a public cloud based on this project. This may be due to licensing issues, as it is a proprietary project, and therefore, it is not easy to modify its core to cater to specific company needs.

Another reason for the lack of a public PaaS based on Aneka can be that this project does not actually show off a cloud usage scenario on its web page, instead it publicizes its usage on batch oriented parallel computing, a workload that is more akin to grid computing.

In [25], authors present performance results after integrating Aneka's PaaS into Azure. The results present an increase of throughput with each increase of working nodes. However, the throughput increase presented is not linear and the performance with 16 working nodes is roughly 68 % lower when compared with the theoretical throughput of 16 nodes, given the throughput of one node.

2.3.2 Google's AppEngine

Google AppEngine (GAE) [26] supports Google approved subsets of Python and Java programming languages, along with specific Google AppEngine APIs. It also features an impressive structured storage solution. AppEngine's data storage solution, BigTable, when accessed from a Java program, features an interface similar to the Java Persistence API (JPA). It also supports a query language similar to SQL but without some of the complex queries, deemed google query language (GQL) [27, 28].

2.3.3 AppScale

An open source project, deemed AppScale, tries to replicate the functionality of GAE and interact with existing IaaS layers [29, 30]. This is one of the projects that most resembles the implementation described herein, since it is also deployable on common IaaS and features an application server (HTTP and HTTPS) based on a servlet container.

This implementation started as a way to deploy the open source AppEngine's SDK over existing IaaS, but over time it grew into a fully featured PaaS. It currently supports most of the functionality of the Google service, and uses a variety of infrastructure resources that can range from physical servers to IaaS.

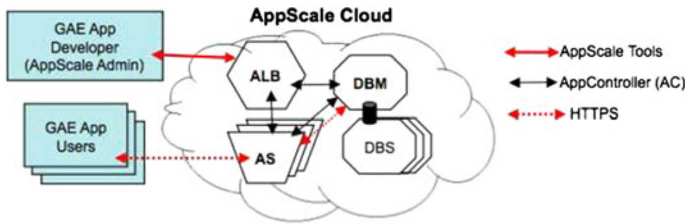

Fig. 4 AppScale design [29]

In terms of architecture, as illustrated by Fig. 4, each AppScale deployment features several modules: an AppLoadBalancer (ALB), one or more AppServers (AS), one DatabaseMaster (DM) and one or more DatabaseSlave (DBS). Virtual machine (VM) nodes execute these modules independently or in any combination of modules. Each VM includes an AppController (AC) that starts automatically on boot. The Head Node (where the ALB is being executed) is responsible for deploying and booting the required additional nodes, and for their configuration in terms of running modules. Along with configuration duties, the AC is also in charge of monitoring the health and status of each node. If required, the AC may commission and decommission nodes to better adjust to the desired application's behavior and/or cost goals.

As mentioned, the ALB is unique within each AppScale deployment. Its main responsibility is to route users to an AS. After the initial routing, users are permanently routed to the given AS. If the AS becomes unreachable, then the user will again be routed through the ALB to a new AS.

The AppServer extends the GAE's development server. The functionality extensions range from a fully automated execution of GAE applications to generic data store interfaces capable of Put, Get, Query, and Delete operations.

2.3.4 OpenShift

The JBoss community started the PaaS development with the cooling tower (CT) [31], advertising it as a turnkey PaaS solution. This project was based on a traditional JBoss application server cluster and on the automatic deployment of the required software stacks to scale the cluster.

Cooling tower claimed to be a build once and deploy anywhere type; quite similar to the objective of the work presented here. This project is still at an early stage, but it already features an interesting persistence layer, Infinispan [32]. Recommend for application data storage, Infinispan is a persistent extension to the java.util.Map interface that allows a simple data model. It also features very interesting core features such as automatic management of data copies and data balancing among available nodes.

This project should allow the usage of most of the J2EE specifications, with exceptions for every part of the standard that relies on the existence of a relational database for persistence, such as the JPA.

Application wise, CT presents a single deployment environment for several running instances. It features an internal Naming Server, where all the accessible running instances must be registered.

Cooling tower, with fundings by RedHat, later evolved into SteamCannon. By the end of 2011, RedHat announced its new PaaS solution based on the recently acquired Mankara: OpenShift [33].

OpenShift claims to be completely compliant with the EE6 standard. Architecture wise, OpenShift is divided into Gears, Nodes, and Districts, as depicted by Fig. 5.

Each Gear enables the execution of one or more applications, named Cartridges. A cartridge is an abstraction for both language based applications (like PHP, Ruby, etc.) and database applications (like PostgreSQL, MySQL and MongoDB). Each Gear runs on a single VM, named Node. A District is a bundling of Nodes within which Gears can be moved freely to better load-balance execution requests.

The high level of abstraction in the cartridge module allows OpenShift to present itself as non-locking-in PaaS solution where running applications on this PaaS will function normally and without developers' intervention on more traditional web servers. In addition, applications currently deployed on traditional web servers can be integrated into this PaaS seamlessly.

2.3.5 WSO2 Stratos

WSO2's Stratos service [34, 35] is a traditional PaaS with an open stack that is deployable over any IaaS or physical infrastructure. It takes the form of a middleware platform that eases the process of building cloud-enabled software. This platform claims to help developers avoid the vendor lock in problems. The approach is somehow similar to OpenShift where WSO2 Stratos provides "Carbon as a Service", with Carbon as a component-based framework for building SOA servers. Web Hosting, logging, clustering, caching, etc. are deployed in the form of Carbon components. The composition and grouping of these components by end users defines a server (a product).

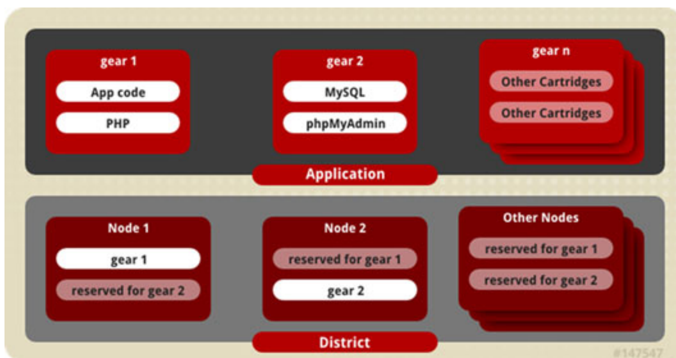

Fig. 5 OpenShift resources and containers

This PaaS supports multi-tenancy with full logical isolation on both storage and execution components. Since tenants are completely isolated, scalability is achieved by scaling the execution components of the application and statically partitioning tenants by the newly created resources.

2.3.6 COSCA

COSCA [36] presents a component-based PaaS solution for common applications. Running on top of the JVM, it supports heterogeneous configurations and combinations of applications running on each executing node. It supports elastic and automatic scaling, virtual addressing of applications, resource consumption-based cost tracking and multi-tenancy. It also claims to be a cloud-independent programming model, avoiding the vendor lock-in effect.

COSCA is still in an early stage and unable to handle and integrate distributed application states.

2.3.7 BeanStalk

Amazon's BeanStalk [37] provides PaaS access to Amazon's resources by means of SDKs for each runnable software stack: Apache HTTP server for PHP and Python, IIS 7.5 for .NET, and Apache Tomcat for Java. BeanStalk presents itself as an "*easy to begin, impossible to outgrow*" solution. It supports elastic and automatic scaling of deployed applications. However, the request for an amount of resources over 500 applications or 20 Amazon EC2 instances requires previous manual authorization.

2.3.8 CloudBees

CloudBees [38] presents itself as a multi-cloud, independent PaaS, capable of operating on simultaneously on multi-cloud environments, on and off-premises. It does so by building on top of the IaaS layer, which has a more stable interoperability. Capable of running any JVM based framework, it relies on its DEV@cloud development environment and the RUN@cloud runtime environment. For helping developers to be in control of the entire applications' lifecycle, they provide Jenkins support.

2.3.9 Cloud Foundry

VMWare Cloud Foundry [39] claims to be an open PaaS, due to its open source nature. Cloud Foundry holds different concepts: a publicly available PaaS for running applications (operated by VMWare), a Micro Cloud Foundry (usually used for testing applications before launching them into the PaaS), and an open-source project and community. Development bases on "standard Cloud Foundry tools" and services, meaning that even though not locked to a provider (due to its open source nature), developers are still locked in the developing model. It currently supports Spring, Ruby on Rails, Ruby and Sinatra, Node.js and Grails as development

frameworks. Similar to CloudBees, Cloud Foundry is an independent PaaS capable on running in several different IaaS providers.

2.3.10 Cloudify

GigaSpaces Cloudify [40] is an open Source PaaS that claims to enable the moving of applications to the cloud with no code changes and, of course, no lock in. Similar to Carbon as a Service from WSO2 Stratos and OpenShift's cartridges, Cloudify claims to support "any app, any stack" on virtually any cloud and using multiple cloud providers at the same time. For the time being, Azure and EC2 are integrated and Apache CloudStack, OpenStack, Rackspace, GoGrid, and Citrix XenServer are on the way. For ease of development, Cloudify provides a fully functional emulator for desktop experiments.

2.3.11 Cumulogic

Cumulogic [41] presents a PaaS ready for integration over HP Cloud Services, Apache CloudStack, OpenStack and Eucalyptus. It provides a standard Java based development model, therefore claiming to prevent vendor lock-in.

2.3.12 EngineYard

Engine Yard [42] is a PaaS that runs on top of Amazon EC2, based on open source technologies. It provides support for Ruby on Rails, PHP, and Node.js.

2.3.13 PaaS Compared

From the above-mentioned PaaS solutions, AppScale, OpenShift, WSO2 Stratos, COSCA, Cloudify, Cumulogic, and EngineYard try to solve the provider lock in issues that arise with PaaS vendors. Interestingly, except for EngineYard (with no source code available) and Cumulogic (only for qualified Universities), all other solutions are based on open source projects. Even if one of these vendors would decide to abandon the project, and there were PaaS restrictions involved, users would not need to migrate to another PaaS right away (but would need to deploy this PaaS elsewhere compatible), which at least alleviates the vendor lock-in issue.

OpenShift, COSCA, and Cloudify claim to be a truly transparent PaaS, with no code changes or special care required for deploying applications on the cloud, meaning that a running web application currently on a traditional server could be easily migrated and benefit from the PaaS infrastructure. The remaining solutions require some sort of modified build environment to build for the cloud runtime.

Each solution presents a different architectural structure, even though the underlining concepts are equivalent: load balancers, master registries and execution nodes (of system tasks or user tasks).

3 Zooming in on the PaaS

This paper presents the efforts taken to develop a fully transparent PaaS runtime that requires no special SDK. The only foreseen restriction is the locking of some package versions due to container implementation restrictions. The design priority for this PaaS is to ensure that developers have to make as little modifications as possible to the standard development practices and patterns that the modern development strategies already recommend.

Traditional web server clusters include a load-balancing layer, an application server layer, and a data persistence layer. Depending on the implementation model of these clusters, they may present a scaling bottleneck, as each of the servers must synchronize its state with the remaining ones, creating a complex web of connections that limit the growth of the cluster. In our approach, this problem is tackled by introducing some modifications on the container, as well as two restrictions on the programming model: no direct access to the filesystem and no access to the server context.

One of the initial goals for the PaaS was its ability to run fully functional unmodified open source web applications. To do so, the PaaS requires a web server that can behave seamlessly as a web cluster. Figure 6 illustrates the proposed PaaS architecture. Technically, the proposed architecture is a *near cluster*, where each application container can host a different set of applications. The application state is stored centrally after serving each request. Extra work is placed on the load balancer, which needs to examine the requests and forward them to an appropriate application server. When deployed, each application server may hold a different set of applications. It is then up to the load balancer to route the incoming requests to the proper server. The proposed architecture has the same layers of a traditional web cluster, but adds a different scaling model, similar to the underlining concepts depicted at the end of the last section, enabling each of the deployed applications to scale independently of the remaining ones.

As explained in [12], one of the proof of concepts involving this PaaS would be its ability to run on operating systems and hardware out of the cloud manager's control, refurbishing concepts as volunteer desktop computing. As such, Fig. 6 shows the involvement of users on a "user provided node configuration". This configuration step enables volunteer node owners to present their resources to our PaaS pool of resources. After the registration phase, when the new node registers its computing availability to the management services, its resources can be used to host applications. Along with this node harvesting model, more traditional (PaaS controlled) spawning models are also available. Within these models, newly deployed virtual machines register themselves in the PaaS pool of resources.

With the proposed architecture, the traditional web server cluster's main components (the load balancer, the application server and the storage backend) continue to be available and operate as expected from an external point of view. This allows applications to run mostly unmodified.

The architecture's component nodes or building blocks can be roughly divided in three distinct categories: management, processing, and support.

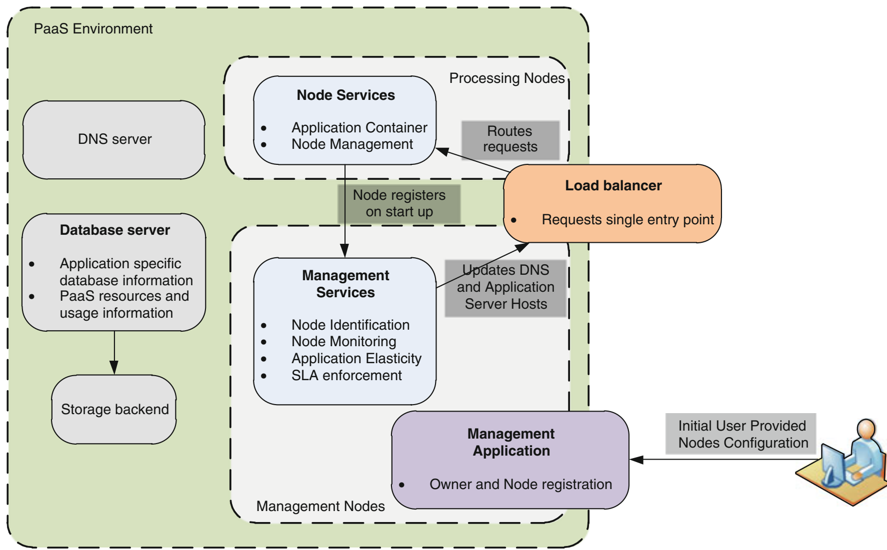

Fig. 6 Proposed PaaS architecture

3.1 Management Nodes

Management nodes write to the main cloud management databases and run one or more of the components described below.

3.1.1 The Management Application

This application is responsible for the management of the users and their nodes. The application manages the main database and allows the visualization of the cloud status as a whole. Besides the load balancers, this is the only visible part of the cloud accessible from outside. Being a web application, it can also be used as a concept showcase, since all the design and execution principles of a web application hosted on a cloud are in place.

3.1.2 Management Services

This is a set of web services run by the management nodes. These services perform the identification and monitoring of the processing nodes as well as the management of the application's elasticity and SLA enforcement. Application elasticity depends on various factors, such as the load of the nodes on which the application is deployed, the application's SLA, the billing information, or even the limits and thresholds set by the user. When needed, a given application may be deployed on an extra node. Each scaling operation consists of assigning or de-assigning a given application/host pair. Upon each scaling operation, the application/host information on the central database is updated to ensure a proper load balancing operation. An application deploy operation depends on the management service to actively

communicate with an available node and send it a new application package to deploy on the web server. The application uninstall procedure depends on ordering the node's web server to uninstall the given application.

3.2 Processing Nodes

Processing nodes are the main worker nodes. These nodes run a modified web server instance, along with the management application described below. Within the web server instance, each node may deploy an assorted number of virtual hosts, one for each deployed application. If required, applications may store and retrieve information from a central application database.

Each node hosts a set of services that allow monitoring and secure deployment of applications. These services run on the web server instance and store all sessions on a central database. For security measures, access to these services is SSL protected, requiring the authentication of the invoking client.

3.3 Supporting Services

Load balancing in this scenario is complex, since not all the application servers host the same set of applications. This means that the load balancer must be designed in terms of sets of application/host instead of simple host availability. The application/host availability information is acquired from a central database. All of the incoming traffic routes through the load balancer.

For security reasons, a DNS server is required to ensure the SSL based establishment of the trust relations between the processing nodes and the management nodes. It is imperative that both nodes have their DNS entries updated before attempting to perform the SSL handshake.

Almost every component of this PaaS uses a database. For convenience, an SQL server was used throughout this project, but a close analysis of the data stored on each database may prove that some of the data are better suited for storage on other types of databases. Examples of such data are the HTTP sessions that can easily be migrated to a key-value store such as redis [43].

3.4 Architectural Issues

One security issue present in this architecture is the possibility of attacks against the central session database. On a final product, the database containing the session data should be replaced, and most likely, an abstraction layer should be used to store and retrieve sessions from the central repository.

There is no sharing of the application server's context among instances. This is probably easy to solve by storing the context properties along with the session on the central database. No attempt was made to do so since the session is usually the only place used to store application state.

Storage is also not shared among instances. This requires that the applications are aware of that, and do not try to store files locally. Some practices, such as creating a cached version of the html output for a file may prove to be counterproductive in

this scenario as such a cache would exist on a single node and could be invalidated by updates performed on a different node. The same is valid for user uploads, which would be saved to a single node, and therefore would not be accessible from other nodes. This can be solved by using a simple storage service that allows a distributed storage to be seamlessly integrated with the application. However, that was not the scope of this project.

Concerning latency, each request answered on this platform involves a few mandatory round trips to one or more databases. The load balancer performs at least one request, and another request originates from the application to fetch any session information required to serve the request (even if the application itself requires no data from the database). As long as the database can perform, latency is expected to be constant regardless of the number of running nodes and application requests.

Finally, all the application requests are bound to go through the load balancer. When on heavy loads, this may represent a bottleneck. Even on more conservative load usage, if the load balancer becomes unresponsive, the entire solution comes to a halt. To better deal with this unresponsive load balancer, redundant load balancer nodes could be implemented. To better deal with heavy loads, another possible solution would be the deployment of a set of load balancers per application, thus minimizing the overall risks of load balancer failure.

3.5 Architectural Benefits

Despite some architectural issues, there are also some benefits from such architecture. The most important is the central session storage that does not suffer from the problems that replicated sessions usually suffer, e.g., a high level of chatter between the cluster nodes or missing some session data if the data is modified directly.

Regardless of the session's centralization contribution to latency, storing the session always after serving a request (or a set of requests) is a simple solution to a complicated problem. Traditional replication techniques require that the server nodes communicate with each other. This architecture removes this requirement as the only time a session is requested from central storage is when one of the processing nodes needs to access that session to output the HTTP response. This enables the cluster size to grow far past the point where the traditional cluster container starts spending more time coordinating than serving requests.

In terms of scalability, the proposed architecture should be able to scale applications horizontally. In an application server cluster, the number of requests answered should scale almost linearly with the addition of more nodes to the pool of nodes running the same application. Since this architecture is based on the same principles of the traditional clusters, this rule should also apply. This potential benefit will be monitored closely during the performance tests.

Considering the developed PaaS model, there is no need to change or rewrite code that already runs on a standard web server. The only exception would be the use of local resources, as explained previously. Even so, this model ensures no vendor lock-in.

3.6 Comparison with the State of the Art

Despite clear differences in terms of maturity levels, an architectural comparison between the proposed solution and the most relevant PaaS presented in the state of the art is mandatory.

As most of the solutions presented, we propose the sharing of execution servers by several applications. It is up to the overall system to minimize the running execution servers. OpenShift, WSO2 Stratos, COSCA, Cloudify and Cumulogic also present a shared environment along with a truly vendor lock-in free PaaS solution, just as our proposal does.

As with AppScale, COSCA and most PaaS solutions, we enable the commission and decommission of execution nodes on demand.

The proposed PaaS solution is not as generic as the Carbon based model presented by WSO2 Stratos or as the cartridges model of OpenShift. It is, in fact very similar to COSCA's proposal. Despite that, it holds pretty much the same components and possibilities as the rest of the pack, even though in a less flexible composition approach. This is related with the initial goal set: to provide a PaaS capable of deploying web applications that were not developed for the cloud. In such a scenario, most of the presented PaaS flexibility is not required at the application level, being something handled during deployment.

Another important goal is related to the previous work presented in [11, 13], where we discuss the incorporation of off-premises computers (personal computers or servers—user nodes) into the cloud's resources to be managed by the PaaS. In this case, the user nodes are incorporated (in some extent, user configured) into the cloud's available resources and may be used to host and run third party web applications. Therefore, this PaaS has to handle the process of registering the user nodes into the available resource pool, by means of a secure automatic enrollment process initiated by the execution node. This idea resembles (for its use of off-premises resources) the Program Ultra-Dispatcher (PROUD) approach. PROUD [44], despite not being a PaaS solution, clearly provides the means for users to take advantage of both the clouds' resources and underused resources of nearby computers. However, and contrary to our concept, these resources are controlled (and owned) by the user; regardless of being physical computers or virtual computers. The PROUD focuses on enabling a given user to execute their applications on the most convenient location, based on preset rules that are based on computer CPU usage, memory usage, etc. The results of running the application are seen just as if they were being executed locally, thanks to the feedback of the remote desktop. In our approach, we provide the PaaS and computing nodes (regardless of the nodes' actual ownership) to support the developers' applications. The typical output is only seen using a web browser to access the deployed application.

Finally, in terms of cloud usage, our approach enables sky computing usage, along with the user node providers.

Table 1 provides a brief comparison among the different PaaS presented in the state of the art and our approach.

Table 1 PaaS compared with state of the art

	Vendor lock-in	Open source	Main deployment language	Sharing of execution servers by applications	Commission and decommission of executing nodes on-demand	Use idle computer resources	Secure enrollment process for executing nodes
Manjrasoft's Aneka	YES		.Net				
Google's AppEngine	YES	YES	Pyhton, Java		YES		
AppScale		YES	Python, Java		YES		
OpenShift		YES	Node.js, Ruby, Python, PHP, Perl, Java and others	YES	YES		
WSO2 Stratos		YES	Java	YES	YES		
COSCA			Java	YES	YES		
BeanStalk	YES		PHP, Python, .NET, Java		YES		
Cloudbees	YES		Java, JRuby, Grails	YES	YES		
Cloud Foundry	YES	YES	Spring, Ruby on Rails, Ruby and Sinatra, Node.js, Grails	YES	YES		
Cloudify		YES	Any app, any stack	YES	YES		
Cumulogic		EDU	Java, Spring	YES	YES		
EngineYard			Ruby on Rails, PHP, Node.js		YES		
Our proposal			Java	YES	YES	YES	YES

4 The Implementation

Java was used for most of the programming needs, despite the SOA based architecture and intrinsic service independency (in terms of service hosting platforms and programming language).

After breaking down the inner workings of a typical web server farm, the conclusion is that most of the required components are already in place and working properly. The requests' distribution mechanism is mostly in place, and the current application containers already support central session storage but do not use it for every request. The missing pieces are the autonomics, the load balancer, and the accounting [13]. All of the components of the architecture depicted on Fig. 6 are operational, even though some improvements are in order.

For the time being, the load balancer is an unmodified HAProxy. This means that the Load balancer is capable of balancing requests only on a node basis. For the purpose of this paper, this does not pose any kind of limitation, since all tests will focus on a single application. However, to support multiple application/host features, the HAProxy balancing rules must be redesigned.

The DNS server is an unmodified PowerDNS application.

The platform data, the application data, and the session data are all stored centrally. An extensive analysis of the available data storage possibilities is required in order to optimize it. However, for now, all data is stored on a MySQL node instance for its simplicity and availability.

For the application container web server, we selected the Apache Tomcat server. The Tomcat server was chosen for its simple architecture, plugin model and ease of changes in its core functionality. Application isolation is achieved via Tomcat's virtual hosting features. If needed, further security measures may be enabled via the container's security manager. The Tomcat server was modified to load the http session information from a MySQL database when a new request arrives, and store it back on the database when it is no longer needed. This allows a series of standalone Tomcat instances that have the same (and unique) application deployed to behave very similarly to a cluster. It also allows a group of heterogeneous instances to behave as a series of clusters, as if each was running one application.

For simplicity, no session locking is performed, meaning that concurrent requests that use the same session, but are directed to different application servers, may end up on storing an unexpected (although consistent) session state. With concurrent session writes, only one of the information sets will be stored. The session locking was left out for simplicity, and in our preliminary testing, no session contention problems were found. Two assumptions justify the decision:

1. Typically, http session dependent applications (stateful applications) are user directed. Users will mostly use browsers that support http 1.1 and a persistent connection, and the system will not store a session to the database until all the requests that use it are served. Even so, as with shopping carts and other web applications, on some circumstances, e.g. if the application server is sluggish or unresponsive, prior to the reception of the HTTP Response, users may issue a new (identical) HTTP Request. With no session locking, if the request alters the session information, this will result in keeping session changes only of one of the requests, typically the later. Under normal user usage (HTTP Request/Response), the session is updated before the new HTTP Request is issued.

2. On the other end, stateless web applications do not depend at all on the session, or depend only on a session set up step, with subsequent requests behaving as read-only from the http session point of view.

The management application is the user interface that allows the cloud supplier to manage the whole PaaS. This is an ongoing project. This application is built using a common set of technologies that can be found on modern Java based web applications (STRUTS2/JPA2). In its current deployment, it manages and monitors part of the cloud's central database. Being a regular web application that was not developed to run on the cloud, it is a good candidate to deploy over the proposed PaaS solution. Figure 7 presents a preliminary look and feel of the management application.

The management services register available processing nodes, communicate securely with the registered nodes, monitor them, and manage their deployed applications. To perform such a list of tasks, these services also manipulate the internal DNS server records, allowing the update of node entries without a static IP address.

Node management services expose methods that allow local resource monitoring and the upload of standard application packages, such as a WAR. Upon upload, the service configures a new virtual host on the Tomcat server to host the uploaded application.

During the development of the platform, some hard to solve problems were faced. Things like which container to choose for modification or how to monitor the physical host running the JVM, were not trivial at that time and took quite a bit of experimenting.

For the application container, the initial choice was to modify a full J2EE container, to seamlessly supply the full power of J2EE as a cloud platform. However the code base of such a project is huge, and takes quite a lot of time just to understand how the several modules work and how the codebase is organized. After a bit of struggling with the GlassFish code base, a servlet container was chosen as a

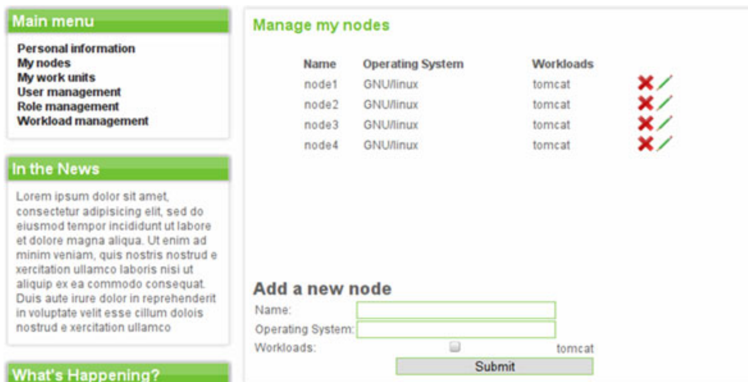

Fig. 7 The management application

subset that would validate the concept and still run many different applications. Tomcat was chosen over Jetty due to background knowledge on its administration.

5 Testing

Testing the functionality of the proposed architecture allows the validation of the architecture's reliability. Testing will also provide some performance and scalability analysis.

5.1 Testbed Description

The testbed runs in a group of physical machines each one with one or more virtual machines. The physical machines run Scientific Linux 5.5, with the Xen hypervisor, and work as virtualization nodes for an OpenNebula based IaaS. The virtual machines run the last beta release of Red Hat Enterprise Linux Version 6 (RHEL 6).

During the initial testing phase, the available hardware was two "long-past-their-half-life-expectancy" servers. By using aged hardware and proving that this PaaS scenario could function on this configuration, we could take advantage of this obsolete equipment to perform tasks that they were no longer able to perform. The best server (Server 1) is a Dell PowerEdge 2850 with two Xeon CPU's @2.8 GHz with hyperthreading, 2 GB of ECC RAM, and 33 GB of disk space. This host runs its software on 250 MB of RAM, the remainder is left for the virtual machines. The other server (Server 2) is a Dell PowerEdge 4400 with two Pentium III Coppermine@800 MHz CPUs, 1 GB of RAM with only 250 MB for the system and about 100 GB of storage. Server 1's disk space was used for operating system images' storage. Server 2's storage is composed of slow disk space that was used for backups and source code repositories.

The tests performed with this hardware configuration did not show a linear relation with the increase of the number of nodes. Given the two physical server deployments with a high number of virtual machines running from the same disk, the observed bottleneck was both on CPU and on the heavy IO performed on the disk and network interface on the same host. In fact, all of the hosts (virtual and physical) were swapping out part of their main memory, and all the swap files and partitions were on the same disk.

This minimal hardware setup enabled us to prove that there is no need for high-end hardware requirements to successfully validate a PaaS architecture under low and moderate loads. However, the original goal was to assess the scalability and performance of the PaaS under heavy loads.

Considering the findings in [30], regarding the similar results of executing tests on physical machines or on virtual machines running on the same physical host, we set up a new hardware testbed with a single physical host: a twin six-core Intel Xeon @2,667 GHz CPU with hyperthreading and 98 GB of ECC RAM, backed by a 1 TB fibre channel storage unit.

The testing was performed over a set of 18 virtual machines, each running Scientific Linux 6. One of the VMs was set aside to run the management services,

the MySQL database, and the load balancer, while the remaining VMs were left as processing nodes. Each VM was allocated one CPU core and 4 GB of memory. All virtual machines were running SNMP and were monitored by an external system while the tests were performed. The virtual machines were managed using OpenNebula.

The setup environment uses NAT networking instead of bridged networking. Because of the NAT networking, the virtual instances are not accessible from the outside world. However, such access is not required; the only service that absolutely needs to be accessible from the outside world is the load manager that may run directly on one of the physical hosts.

Throughout the testing, one virtual machine instance was used to run all of the management services and the other instances were reserved to be used as processing nodes. The virtual machines were kept running throughout the whole test, but the services that composed the processing node were only started on an "as needed" basis.

As mentioned before, the Management Application will be used for testing purposes.

5.2 Reliability

The reliability test results will indicate if the service is able to function for low loads, as long as at least one processing node runs the target application. This test started with the management node running, and two processing nodes running the same application on the same named virtual host. After deploying the application, the application code was manually modified (on each virtual host) to identify which of the hosts responded to a request. This helped on reinsuring that the requests were not all being sent to the same node.

A custom entry was added to the hosts' file on a client desktop pointing to the selected name for the virtual host to the IP address of the load balancer. The application was then accessed using the laptop's browser. The navigation was as follows: first, login as administrator; second, click once on each menu entry; third, update the user name; forth, logout.

This navigation sequence was repeated with different amounts of running nodes. Each sequence always worked as expected, showing little or no session stickiness, proving that the load balancer was not configured to stick each session to a server instance, and that the modifications made to the Tomcat server work at least under a light load. The navigation sequence was recorded using JMeter, which was later used to test the performance and scalability of the system.

Additionally, one of the nodes was shut down, leaving only one of the initial nodes running. A new node was booted, the application was deployed on that node and it was added to the load balancer. Finally, the remaining original node was shut down, leaving none of the original nodes running. As long as at least one node is running the requested application, the system will keep the application state.

5.3 Performance

The performance tests were made using JMeter-2.0.1, running on a different physical computer. It simulated the workflow described above for a consecutively higher number of users (threads). Each thread executes the workflow 20 times. Each workflow consists of 13 requests, 11 of which require application database access and one is a static image. JMeter was configured with a Gaussian timer with a 300 ms delay and 100 ms of deviation.

We set a maximum response time on 500 ms and tracked individual response times. For each number of active nodes we calculated the 90th percentile of the time of the response for a given amount of threads. The usage of a high percentile instead of the simple average enables us to best cope with network fluctuations and other random abnormalities. By plotting the information of each amount of threads per each number of nodes, we calculated the real number for the 90th percentile for each number of nodes. Figure 8 summarizes results along with a trend line and associated equation.

Analyzing the number of threads per number of nodes, there is a performance cost of roughly 2.5 % per increased node, adding about 30 % of theoretical performance loss when comparing the throughput of 3 nodes with 17 nodes.

An analysis of the monitoring data collected while testing shows that there is a possible performance bottleneck on the PaaS: the MySQL server. With the higher numbers of processing nodes, MySQL was using about 80–90 % of its available processing power. This behavior is easily explained by the amount of database requests being made: at least one for each request (caused by the application state being stored in a database) and on most cases another access for application specific data.

Despite the overall 30 % performance loss reported, the results are encouraging when compared with others: Aneka [25] running on Windows Azure reported about a 68 % theoretical performance loss on 16 nodes, which translates into a 47.4 %

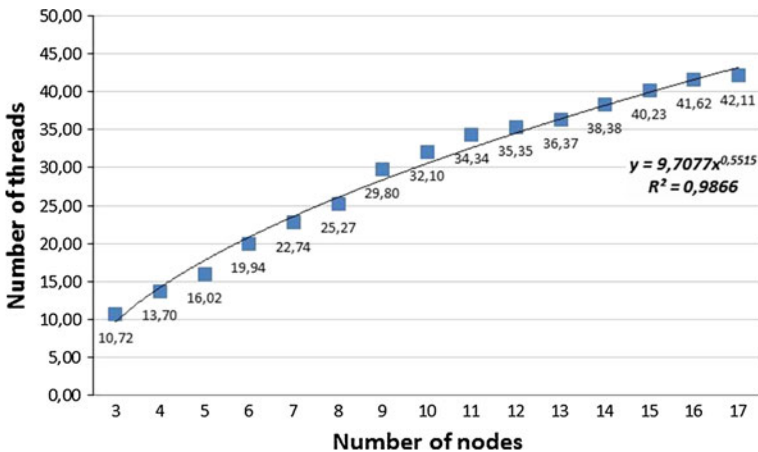

Fig. 8 Threads for 90th percentile requests below 500 ms

loss when scaling from 4 to 16 nodes. With our approach, the same scaling (from 4 to 16 nodes) sets the performance back by 27.6 %.

6 Conclusions

We were able to successfully run a fairly complex Java based application in a distributed web environment, with no specific code change using our developed PaaS. The only foreseeable restriction is the usage of the processing node's local resources (no local paths should be used). This web application was developed using an ordinary version of NetBeans, as a regular web application. Despite the lack of knowledge of the cloud's insights, the web application can be deployed in the cloud environment where it can be scaled up and down according to the application's needs. All the cloud's insights and inner works are completely transparent to developers.

The performance results show a very well behaved relation with the increase of the number of nodes, which allows us to safely state that the PaaS solution proposed presents a good scalability solution. One that shows better results than more mature (and of course more complex) available PaaS solutions.

Despite the good overall performance, the metrics gathered from the servers via SNMP raised a yellow flag concerning the MySQL server resource consumption. As previously mentioned, a thorough investigation of possible database solutions is mandatory at this point.

Even though there are still loose ends and issues to solve, at this stage it is already possible to assess the viability of the proposed solution. The final architecture displays some interesting strong points even when compared with the state of the art PaaS solutions.

To take advantage of the platform's elasticity, the load balancer needs to be implemented from scratch because none of the evaluated load balancers has the features or the code organization that would allow a simple modification to fulfill the requirements of our PaaS.

To test this PaaS to its full extension, the next step will be to deploy it on a sky computing approach, trying to understand the benefits (if any) of a heterogeneous solution.

Open Access This article is distributed under the terms of the Creative Commons Attribution License which permits any use, distribution, and reproduction in any medium, provided the original author(s) and the source are credited.

References

1. Pallis, G.: Cloud computing the new frontier of internet computing. IEEE Internet Comput. **14**, 70–73 (2010)
2. Grossman, R.L.: The case for cloud computing. Inf. Technol. Prof. **11**, 23–27 (2009)
3. Borenstein, N., Blake, J.: Cloud computing standards where's the beef? IEEE Internet Comput. **15**, 74–78 (2011)

4. cloud-standards.org.: Cloud standards overview. Available: http://cloud-standards.org/wiki/index.php?title=Cloud_standards_overview (2010, 13/06/2011)
5. Smith, D.M.: Hype cycle for cloud computing, Gartner2011 (2011)
6. Mell, P., Grance, T.: NIST definition of cloud computing (Draft). NIST (ed.) (2011)
7. Höfer, C.N., Karagiannis, G.: Cloud computing services: taxonomy and comparison. J. Internet Serv. Appl. **2**, 14 (2011)
8. Apache. Deltacloud project. Available: http://incubator.apache.org/deltacloud/ (2011, June 14)
9. Red Hat Inc. Aeolus project—manage your cloud deployments with ease. Available: http://aeolusproject.org/ (2011, June 14)
10. O'Reilly, T.: Is Google App Engine a lock-in play?—O'Reilly Radar. Available: http://radar.oreilly.com/2008/04/is-google-app-engine-a-lockin.html (2008, June 14)
11. Teixeira, C., Azevedo, R., Pinto, J.S., Batista, T.: A cloud oriented approach for people-provided services. Presented at the Fifth International Conference on Internet and Web Applications and Services (ICIW 2010), Barcelona, Spain (2010)
12. Teixeira, C., Azevedo, R., Pinto, J.S., Batista, T.: User Provided Cloud Computing, Presented at the IEEE/ACM International Symposium on Cluster Computing and the Grid. Birsbane, Australia (2010)
13. Batista, T.S.: Architectures for cloud computing based information systems. Master. Departamento de Electrónica Telecomunicações e Informática, Universidade de Aveiro, Aveiro (2010)
14. Turner, M., Budgen, D., Brereton, P.: Turning software into a service. Computer **36**, 38–44 (2003)
15. Dubey, A., Wagle, D.: Delivering software as a servic*e,* The McKinsey Quarterly 06/2007 2007
16. Weiss, A.: Computing in the clouds. NetWorker **11**, 16–25 (2007)
17. O'Reilly, T.: Is Google App Engine a lock-in play?—O'Reilly Radar. Available: http://radar.oreilly.com/2008/04/is-google-app-engine-a-lockin.html (2008, October 5, 2009)
18. Keahey, K., Tsugawa, M., Matsunaga, A., Fortes, J.A.B.: Sky computing. IEEE Internet Comput. **13**, 43–51 (2009)
19. Monteiro, A., Pinto, J.S., Teixeira, C., Batista, T.: Sky computing—exploring the aggregated cloud resources—part I, Presented at the CISTI2011—6ª Conferência Ibérica de Sistemas e Tecnologias de Informação Chaves, Portugal (2011)
20. Monteiro, A., Pinto, J.S., Teixeira, C., Batista, T.: Cloud interchangeability—redefining expectations, Presented at the CLOSER 2011—international conference on cloud computing and services science, Noordwijkerhout, The Netherlands (2011)
21. Vecchiola, C., Chu, X., Buyya, R.: Aneka: a software platform for .NET-based cloud computing, In: Gentzsch, W., Grandinetti, L., Joubert, G. (ed.) High Performance and Large Scale Computing, IOS Press, Amsterdam, Netherlands (2009)
22. M. P. Ltd. Manjrasoft. Available: http://www.manjrasoft.com/ (2010, May)
23. Vecchiola, C., Chu, X., Buyya, R.: Aneka: a software platform for .NET-based cloud computing. High Performance & Large Scale Computing, Advances in Parallel Computing. IOS Press, Amsterdam (2009)
24. Chu, X., Nadiminti, K., Jin, C., Venugopal, S., Buyya, R.: Aneka: next-generation enterprise grid platform for e-science and e-business applications, Presented at the IEEE International Conference on e-Science and Grid Computing, Bangalore (2007)
25. Wei, Y., Sukumar, K., Vecchiola, C., Karunamoorthy, D., Buyya, R.: Aneka cloud application platform and its integration with windows azure, CoRR, vol. abs/1103.2590 (2011)
26. Google. Google App Engine. Available: http://code.google.com/intl/pt/appengine/ (2009, 12/2009)
27. Chang, F., Dean, J., Ghemawat, S., Hsieh, W., Wallach, D., Burrows, M., Chandra, T., Fikes, A., Gruber, R.: Bigtable: a distributed storage system for structured data, Presented at the 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI '06), Seattle (2006)
28. Google. GQL reference—Google App Engine—Google code. Available: http://code.google.com/appengine/docs/python/datastore/gqlreference.html (2010, May)
29. Krintz, C., Bunch, C., Chohan, N.: AppScale: open-source platform-as-a-service. In: Vaquero, L.M., Cáceres, J., Hierro, J.J. (eds.) Open Source Cloud Computing Systems: Practices and Paradigms, p. 18. IGI Global, Hershey PA, USA (2012)
30. Chohan, N., Bunch, C., Pang, S., Krintz, C., Mostafa, N., Soman, S., Wolski, R.: AppScale: scalable and open AppEngine application development and deployment. Lect. Notes Inst. Comput. Sci., Soc. Inf. Telecommun. Eng. **34**, 14 (2010)
31. Neale, M. Cooling tower—JBoss community. Available: http://community.jboss.org/wiki/CoolingTower (2010, May)

32. Red Hat Inc. INFINISPAN—open source data grids—JBoss community. Available: http://www.jboss.org/infinispan (2010, May)
33. R. H. Inc. Chapter 1. OpenShift architecture overview. Available: http://docs.redhat.com/docs/en-US/OpenShift/2.0/html/User_Guide/chap-User_Guide-OpenShift_Architecture_Overview.html (2012, 02/05)
34. WSO2. WSO2 Stratos Available: http://wso2.com/cloud/stratos/ (2011, March 29)
35. Azeez, A., Perera, S., Weerawarana, S., Fremantle, P., Uthaiyashankar, S., Abesinghe, S.: WSO2 Stratos: an application stack to support cloud computing. Inf. Technol. **53**, 8 (2011)
36. S. K, #228, chele, #246, r. Domaschka, F. J. Hauck, COSCA: an easy-to-use component-based PaaS cloud system for common applications. Presented at the Proceedings of the First International Workshop on Cloud Computing Platforms, Salzburg, Austria (2011)
37. Amazon. AWS elastic beanstalk. Available: http://aws.amazon.com/elasticbeanstalk/ (2012, 20/09)
38. Cloudbees. CloudBees: The Java PaaS company. Available: http://www.cloudbees.com/ (2012, 07/09)
39. VMWare. Welcome to cloud foundry. Available: http://www.cloudfoundry.com/ (2012, 07/09)
40. Cloudify. Cloudify—the open PaaS stack. Available: http://www.cloudifysource.org/ (2012, 12/09)
41. Cumulogic. CumuLogic—enabling Java platform-as-a-service (PaaS) in public, private and hybrid clouds. Available: http://www.cumulogic.com/ (2012, 21/09)
42. EngineYard. Ruby on rails and PHP cloud hosting PaaS|managed rails development|engine yard platform as a service. Available: http://www.engineyard.com/ (2012, 25/09)
43. redis. redis. Available: http://redis.io/ (2012, 01/06)
44. Huang, T.-C.: Program Ultra-Dispatcher for launching applications in a customization manner on cloud computing. J. Netw. Comput. Appl. **35**, 423–446 (2012)

Author Biographies

Cláudio Teixeira Graduated from the University of Aveiro, Portugal in 2003, with a 5-year degree in Telematics and Computers Engineering. In 2009, he received his Ph.D. degree in Computer Science. He is currently an assistant researcher at the Department of Electronics and Telecommunications and Informatics of the University of Aveiro. His main research interests include integration of information systems, system modeling and architecture, XML-based information systems, cloud computing and e-government.

Joaquim Sousa Pinto graduated from the University of Aveiro, Portugal in 1985, with a 5-year degree in Electronics and Telecommunications Engineering. In 1997, he received his Ph.D. degree in Electronics Engineering. He is currently an Assistant Professor at the Department of Electronics and Telecommunications and Informatics of the University of Aveiro. His main research interests include digital libraries, information systems, and cloud computing.

Ricardo Azevedo graduated from the University of Aveiro, Portugal in 2004, with a 5-year degree in Telematics and Computers Engineering. In 2006, he received his M.Sc. in Internet Computing. He is currently Project Manager in the Innovation Department of Portugal Telecom. His main research interests include identity management, information and network security, privacy, context aware, and cloud computing.

Tiago Batista graduated from the University of Aveiro, Portugal, in 2010, with a M.Sc. on Telematics and Computers Engineering. He is currently working as a Software Engineer in the Innovation Department of Portugal Telecom. His main research interests include identity management, cloud computing, and virtualization.

André Monteiro is currently a Ph.D. student at the Department of Electronics and Telecommunications and Informatics of the University of Aveiro. In 2005, he received his 5-year degree in Telematics and Computers Engineering. His main research interests include cloud computing, virtualization, high performance computing, and service oriented architectures.