A FEDERATED REPOSITORY FOR PAAS COMPONENTS IN A MULTI-CLOUD ENVIRONMENT

Rodrigo García-Carmona[1], Félix Cuadrado[2], Álvaro Navas[1] and Juan Carlos Dueñas[1]
[1]*Departamento de Ingeniería de Sistemas Telemáticos, ETSI Telecomunicación, Universidad Politécnica de Madrid, Madrid, Spain*
[2]*School of Electronic Engineering and Computer Science, Queen Mary University of London, London, U.K.*

Keywords: Platform as a Service, Service Repository, Dependency Resolution, Federation, OSGi.

Abstract: Cloud computing has seen an impressive growth in recent years, with virtualization technologies being massively adopted to create IaaS (Infrastructure as a Service) public and private solutions. Today, the interest is shifting towards the PaaS (Platform as a Service) model, which allows developers to abstract from the execution platform and focus only on the functionality. There are several public PaaS offerings available, but currently no private PaaS solution is ready for production environments. To fill this gap a new solution must be developed. In this paper we present a key element for enabling this model: a cloud repository based on the OSGi component model. The repository stores, manages, provisions and resolves the dependencies of PaaS software components and services. This repository can federate with other repositories located in the same or different clouds, both private and public. This way, dependencies can be fulfilled collaboratively, and new business models can be implemented.

1 INTRODUCTION

Cloud computing has completely changed the perspectives of the execution infrastructure, enabling an unparalleled level of abstraction, as well as the ability to dynamically adapt the system capabilities in response to the perceived demand. This has been promoted by the massive adoption of virtualization technologies, with the leading cloud model being IaaS (Infrastructure as a Service).

However, IaaS offers a low-level approach for cloud developers, which must still manage execution nodes (even if they are virtual), and be concerned with the operation and configuration of lower layers, such as operating systems and application servers.

On the other hand, PaaS (Platform as a Service) solutions promise to allow developers to abstract from the IT infrastructure details and work at application level. Nonetheless, this model is still at its infancy stage, with multiple incompatible solutions being offered, most of them propietary. On top of that, in exchange for the provided abstraction, most of them lock applications into their propietary APIs.

OSGi (OSGi Alliance, 2011) is a component model with ideal characteristics for the development of a private PaaS solution. However, while the specification is mature and has been used for several years in the enterprise application server market, some components and capabilities still need to be developed for its transition to the cloud.

One of them is a component repository that works in a multi-cloud environment. This repository will manage the PaaS software components, store them and enable their correct provisioning through automatic dependency resolution, helped by its capability to federate with other repositories of the same kind..

This paper presents an initial implementation of this repository, detailing its distinctive characteristics. The structure of the article is as follows. Section 2 presents existing PaaS solutions and introduces OSGi as an alternative to them. Section 3 explains what characteristics a cloud repository needs and briefly shows the architecture of our contribution, while section 4 provides more detail over the dependency resolution and federation mechanism. Finally, section 5 closes this paper with a summary of our contributions, and a suggestions of several future lines of work that could be followed.

García-Carmona R., Cuadrado F., Navas Á. and Carlos Dueñas J..
A FEDERATED REPOSITORY FOR PAAS COMPONENTS IN A MULTI-CLOUD ENVIRONMENT.
DOI: 10.5220/0003928004190425
In *Proceedings of the 2nd International Conference on Cloud Computing and Services Science* (CLOSER-2012), pages 419-425
ISBN: 978-989-8565-05-1

2 ELASTIC SERVICE CLOUDS

2.1 PaaS Solutions

The US National Institute of Standards and Technology (NIST) defines PaaS as a cloud computer model where "the capability provided to the consumer is to deploy onto the cloud infrastructure consumer-created or acquired applications, created using programming languages, libraries, services, and tools supported by the provider" (Mell and Grance, 2011). In a PaaS a user can only control what the provider allows him to. Although this limitation could be considered a step back, in reality it enables the user to abstract from unnecessary details, creating an environment where rapid development is a given, as the users' only concern is developing the application.

Most PaaS solutions are public clouds. These are clouds whose infrastructure is open for the general public. Google AppEngine (Zahariev, 2009), Microsoft Windows Azure (Hill et al., 2010) and Amazon's Beanstalk (van Vliet, Paganelli, van Wel and Dowd, 2011) are the most well-known and used representatives of this kind of PaaS. Although public clouds are widely used, they present some drawbacks: the developer must adapt every application deployed onto them to their APIs and special requirements, and security and privacy data concerns can render those solutions unfeasible in some contexts.

On the other hand, private clouds are restricted to the use of selected individuals, normally the same entity that hosts them. Even considering that recent reports (Natis, 2011) see private PaaS as the most interesting future development for clouds, almost every private PaaS cloud project is still in the early beta stage. Cloudfoundry (Wolff, 2011) is the most notorious project. It supports a big array of languages (like Java, Ruby and Scala) and frameworks (like Spring and Lift). However, it is still under development and the support for building private clouds is not complete.

As of this moment, there are no solutions for private PaaS clouds mature enough for being used in production environments.

2.2 OSGi as a PaaS Solution

The OSGi Alliance (Kriens et al., 2011) has already stated why OSGi could be a good addition to the cloud technologies ecosystem. Its modularity and dynamic nature is perfectly suited to the ever changing cloud environments. And in particular, the PaaS paradigm and OSGi look like the perfect fit. A PaaS cloud built upon OSGi does not exist yet, but it has already been proposed, and it should offer the following advantages.

The first advantage is the minor changes that OSGi requires for standard Java applications to work with it. In a PaaS solution the developer usually has to perform extensive modifications in order to make an application work in the new environment. However, a Java application deployed in an OSGi PaaS would require only slight changes. Any framework already supported by OSGi can be used. Moreover, the enforced modularity simplifies the extension of an application in the form of new services, since modules can be easily reused.

Another advantage is that, in an OSGi PaaS, the developer would not have to worry about dependencies. He is expected to only define them, and a cloud repository will resolve and find them, provisioning them to the node in which the application is running transparently. Moreover, available nodes can be used by different applications, significantly lessening the stress on the underlying infrastructure. On top of that, while applying upgrades and roll-backs in a PaaS often requires reboots, an OSGi cloud can be updated on the fly.

All in all, there are several reasons to use an OSGi-powered PaaS cloud, but as this solution is still a work in progress, there are challenges that must be overcome first. Many are directly derived from adapting OSGI to the cloud environment and are common to any platform that wants to act as a PaaS. Provisioning is one of these problems. Bundles are, in fact, the OSGi components, and they enable one of the main advantages of an OSGi environment; its powerful dependency management. Therefore, a cloud bundle repository is a central piece for this model, since it is in charge of resolving dependencies, finding needed components, provisioning them and, in short, enabling the deployment of any application into the cloud.

3 CLOUD COMPONENT REPOSITORY

3.1 Requirements

After having explained how OSGi can serve as the basis of a PaaS cloud, we can now define what requirements a repository should suffice to manage provisioning in the cloud.

First of all, the repository must be able to

manage components, not just store them. Distributed services rely heavily on external dependencies, so dependency management is a core requirement. It is also important that the resolution incorporates multiple compatibility criteria, since an incorrect component closure could leave the system in an unstable state.

As the main source of provisioning in the cloud, the repository must be always on, implementing high availability. Additionally, the repository has to support high scalability, as it must keep working even under great demand. With no repository working no new applications can be deployed.

Finally, the last requirement is derived from the OSGi vision of a PaaS cloud. The cloud provider should be able to offer private repositories to their clients. Each of these repositories can be accessed only by certain users. They must hold any application provided by the specific user, or supposed to be accessible only to him. Additionally, these repositories should also be able to communicate to other repositories to make queries or resolve dependencies. These other repositories provide openly available components (like open source software), which could be used by different users. Hence, the repository must be able to federate with other instances of itself, each one managing a different set of components. Additionally, the repository should be able to federate with other kind of repositories in other clouds. This could be a huge enabler due to the extended service catalogue. However, at this stage of development, it is still an optional requirement.

3.2 Component Repository Technologies

At this moment there are several available component repository technologies. Every one of them has its own component model and capabilities. The most popular ones are detailed in this section. We have put a special focus in their support of OSGi bundles and their federation capabilities, since these are the basic elements needed to fulfil the desired requirements.

Maven (O'Brien, 2008) is one of these solutions, a software project management and comprehension tool. Maven has become the de facto standard for managing Java projects, thanks mainly to the support and its extended use inside the Apache community. However, such a big scope comes with a price, as Maven cannot accurately express the special relationships between OSGi Bundles.

Another model used to describe bundles is the OBR (OSGi Bundle Repository) specification. This model was proposed as the draft OSGi RFC 112 (Kriens and Hall, 2006). Although OBR is the official OSGi solution for the cloud, it is still in a draft stage. There is no OBR client specification and the federation between OBR servers is not well defined either, making OBR a poor solution for the cloud.

In the 3.0 version of Eclipse, the Eclipse architecture was changed to use OSGi as the project core. This change pushed the Eclipse community to develop their own bundle repository, named P2 (Le Berre and Rapicault, 2009). The P2 repository is widely used, as version 3.4 of the Eclipse Platform employs it as the management mechanism for its components (which are, in fact, OSGi bundles). However, this solution has an important drawback: Its component model is concerned only with software direct dependencies, being oblivious to other constraints that could affect artefacts. Since one of the requirements was precisely this, P2 cannot be used as the repository solution. On top of that, P2 needs the Eclipse Extension Registry, making it incompatible with other OSGi implementations.

Finally, there is the OSAmI bundle repository (García-Carmona, Cuadrado, Dueñas and Navas, 2011), which was design keeping in mind concerns similar to the ones exposed before. It offers a software model that captures all the relevant information, a dependency resolution system that is built upon it, and an architecture that can be extended to support multi-cloud federation.

There are a lot of existing solutions for a bundle repository, but this survey shows that no actual repository technology completely fulfils the set of requirements imposed for the implementation of the cloud bundle repository. The one that can be more easily extended is the OSAmI bundle repository. Our cloud bundle repository is built upon its foundations.

3.3 Repository Architecture

In this section we present the architecture for the cloud bundle repository. For its creation we have taken as reference the previously presented OSAmI bundle repository, reengineering it in order to accommodate the requirements emerging from the PaaS cloud, as presented before. For a complete view of the repository architecture we recommend the previous reference and Figure 1, as we will only address the changes to the architecture in this section.

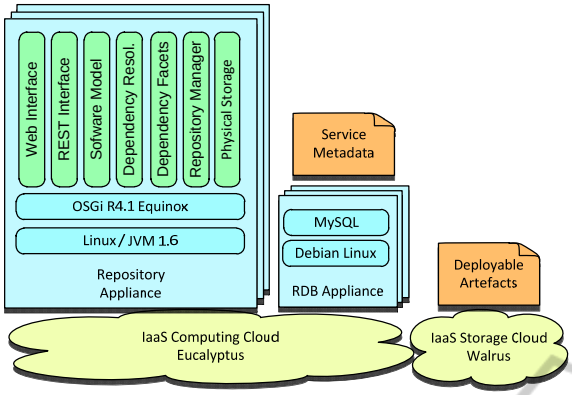

Figure 1: Repository architecture.

The shift from a traditional infrastructure to a private cloud environment does not affect the logical view of the architecture. Instead, it is more interesting to discuss the physical view [4+1] (Kruchten, 1995) of the repository architecture to detail the decisions taken on migrating the service to a private cloud platform. As there are currently no widely available implementations of a private Java PaaS cloud, we chose to deploy the repository on top of a private IaaS. We selected Eucalyptus (Nurmi et al., 2009), an Open Source IaaS technology, as the base cloud. Eucalyptus is a mature infrastructure platform that implements management interfaces compatible with the *defacto* standard in IaaS clouds (Amazon Web Services). This allows us to benefit from the already available third-party management products.

For the logical components (following the OSGi and Spring DM (Rubio, 2009) specifications) we have defined a virtual appliance containing the JVM (Java Virtual Machine), the OSGi framework and all required libraries. The repository components can be directly deployed on top of this platform. The architecture design follows the stateless fundamental principle of SOA systems (Erl, 2005), greatly simplifying the scaling up of the repository through the launch of additional instances. Additionally, a load balancing virtual appliance (composed by a Linux-powered system) spreads the requests between the cloud repository instances.

The shift to a cloud environment impacts persistent data storage the most. However, we have chosen to keep a MySQL database to store metadata as most cloud solutions support the use of relational databases and the expressivity of the SQL language is best suited to perform dependency matching queries.

Finally, the physical storage of the deployable artefacts can take advantage of the scalability and reliability capabilities of cloud storage solutions. We have selected Walrus (An S3-compatible storage cloud, which is part of Eucalyptus), to physically store the files.

4 DISTRIBUTED REPOSITORY

The scenario we have shown in the previous section corresponds to the repository working inside a cloud. But that by itself does not fulfil the requirements specified before. Private clouds must be able to cooperate among themselves, and the PaaS federation level requires as a first step a repository federation network, with the representatives of each cloud cooperating with each other.

It is no longer feasible for a single entity to hold all the required information, all the components needed for a non-trivial application to work. Instead, repositories must be able to federate, relying on each other in order to compose an aggregated service space that can satisfy any requirement that appears in the local domain. Practicality is not the only reason behind this decision, as privacy is an important factor. Not every stakeholder should be able to access each other's components.

This section explains how the repository achieves this federation. First we explain how the resolution mechanism, and the model upon its built, work. Both of them are the foundations of the federation capabilities. Finally, we show how the repository federation mechanism itself works.

4.1 Component Model and Dependency Resolution

The dependency resolution activities performed by the repository are executed automatically when needed, without the human input or supervision. A developer only needs to state the dependencies of his components, and can rely in the repository for finding all the other services and components required. This is enabled by the use of the abstractions expressed in the software model instances.

The central element of this model is the resource. Resources represent any manageable element that contributes to the overall functionality of the whole system. Typical resources include deployed web services, software libraries, stored tables in a database, or available TCP ports from the operating system. A resource is described by a name, a version, and a type. This structure implicitly defines

a taxonomy of managed elements. The resource is further described by a set of properties, formed by simple name/value pairs.

The physical artefacts that are deployed into the system are modelled as deployment units, each a different model instance. A deployment unit is the equivalent of an OSGi bundle. Units provide functionality to the system (expressed by a set of exported resources), and also declare a set of requirements that must be addressed for the unit to work properly. The most typical type of requirement are dependencies, expressing the need for another software resource (identified by its name, type and a range of compatible versions) to be present in the platform in order for the unit to work correctly. Dependencies are the elements that enable the dependency resolution.

Dependency resolution starts with an initial unit or resource being requested. This triggers a recursive search algorithm where the unit dependencies are explored, trying to find compatible units that provide the required resources. In turn, these new units have dependencies themselves. For them to be sorted, this process is repeated for every unit until a complete dependency graph has been formed.

This is a summarised view of the process, as there are additional types of restrictions that can be expressed, both as part of the descriptor (e.g. hardware constraints with minimum requirements or incompatibilities, or general constraints such as license compatibility).

But sometimes this dependency resolution mechanism must be expanded to include other concerns. A dependency resolution process must be able to match a number of criteria not known at design-time. Therefore, the architecture of the repository was defined with the possibility of expanding this mechanism in mind. This is done through components called facets. Each of them provides specific reasoning for second-level concerns, such as special hardware requirements, environmental conditions or software license compatibility.

4.2 Repository Federation

The core of the repository inter-cloud federation lies in the idea of extending the dependency resolution and artefact finding capabilities of the repository to work with other instances running either in the same or other clouds.

This can be seen in the following figure, which depicts a multi-cloud environment, with several private clouds and a public one. Each of them contains at least one cloud bundle repository.

Every repository has a reference that points to at least another repository, in the same cloud or in another one. If this relationship is recursively extended, it can be seen that most repositories are connected to many others, even if a high number of jumps is required. Only the truly private repositories are isolated from the rest. This reference is not bidirectional. This means that the fact that one repository knows about other does not imply that the opposite is true.

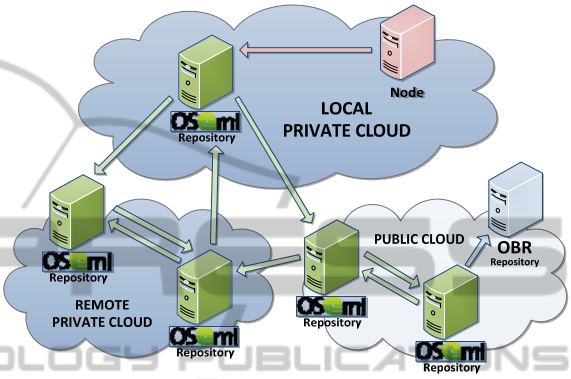

Figure 2: Repository multi-cloud federation.

These relationships represent the capability of a repository to access the components of another, query for its metadata, and ask that repository to perform a dependency resolution. This way, if a repository does not contain the necessary components to fulfil the dependencies of a particular bundle, it could survey other repositories, searching for the missing elements.

To materialize this scenario, the repository needs to be extended with the following elements:

- A remote interface that could be accessed by another repository.
- A client component that uses this interface to remotely access the repository and locally deploy components.
- A dependency resolution algorithm that takes into account this federation.

For the remote communications needs we have developed two modules: a client and a server. They communicate through a remote interface defined using REST, which contains the operations needed for the access, modification, creation and deletion of components, as well as an interface for querying the repository and performing a dependency resolution.

The server is deployed as a bundle within the repository, while the client is used both by repositories and the nodes inside the PaaS. Inside the repository this client serves as a communication module that enables the federation. At the same

time, at the PaaS nodes the client acts as the provisioning service, communicating with one of the local repositories when an order for the deployment of a new component is given.

But these are only enablers. The functionality itself lies in the dependency resolution algorithm, which considers other federated repositories and the components managed by them.

The following picture shows a sequence diagram representing a sample invocation of this algorithm. The actors are the provisioning component, its local repository (with facets), and two remote repositories deployed in an external cloud. One directly accessed by the local repository, and the other indirectly through it.

On top of this algorithm, every repository manages all interactions with other repositories, including the access and download of components stored in federated repositories. The provisioning client will never know of the existence of repositories other than the one he is configured to ask.

5 CONCLUSIONS AND FUTURE WORK

OSGi can be turned into a successful private PaaS solution, but several developments and improvements need to be made before that. The work presented in this article concerns what we consider to be the fundamental one: the definition and implementation of a cloud bundle repository.

Building upon the foundations provided by our previous work in the OSAmI project, we have created a component repository that can fulfil the requirements imposed by working on a multi-cloud environment. One of them, the federation with other repositories running in the same or different clouds, stands above the rest. We have described the components needed to fulfil this requirement and how we have implemented them, including a client application that is responsible for the provisioning inside the cloud.

This multi-cloud federation enables an easier and more organized development and deployment of components. It also opens the possibility of new business models, in which some components are made available only after they have been purchased, and some of them are kept private to some customers. Moreover, this structure enables every stakeholder that desires to maintain their developments private to do that, while at the same time enabling an easy access for its developers and allowed users to additional applications.

Concerning future developments, the work presented in this article could be expanded following two different routes.

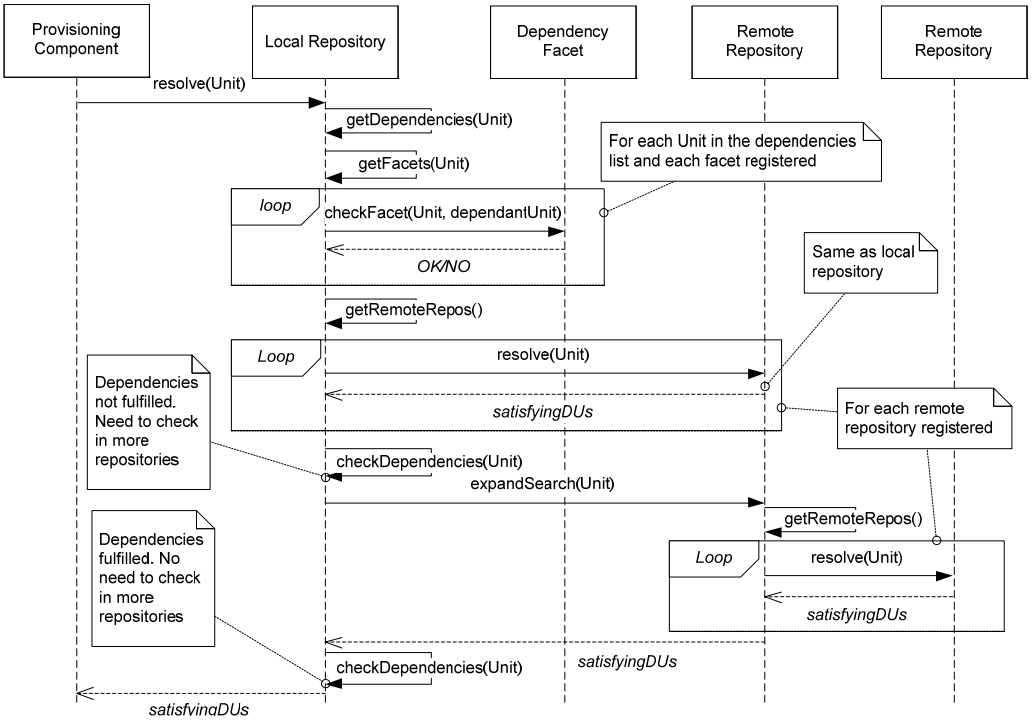

Figure 3: Dependency resolution.

On the one hand, having the cloud bundle repository already available, efforts can be directed to the development of the rest of components and modifications needed to truly turn OSGi into a complete and fully functioning PaaS solution. How to implement logging and monitoring and actual scalability support are just some of the important topics that could be explored.

On the other hand, the federation capabilities of the repository could be also extended. The basic model of the current implementation could be expanded with an intelligent distribution of components, with a cache and replication of the most required bundles into the local cloud or nearest repositories. This will need an advanced set of rules for the distribution of components between different repositories, keeping into account not only their physical capabilities, but also their licensing and distribution limits.

Finally, a set of dependency resolution facets specially designed for cloud concerns, like intra-cloud security, geographical location of the machines, QoS (Quality of Service) characteristics and billing can be integrated into the general architecture of the repository, greatly improving its performance in a federated PaaS environment.

ACKNOWLEDGEMENTS

The work presented in this article has been performed in the context of the European project ITEA-OSAMI, under grant by Spanish Ministerio de Industria, Turismo y Comercio in the PROFIT program.

REFERENCES

García-Carmona, R., Cuadrado, F., Dueñas, J. C., Navas, Á., 2011. A Model-based Repository for Open Source Service and Component Integration. *In: Sixth International Conference on Software and Data Technologies* (ICSOFT).

Hill, Z., Li, J., Mao, M., Ruiz-Alvarez, A., Humphrey, M, 2010. Early observations on the performance of Windows Azure. *In Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing (HPDC '10)*. ACM, New York, NY, USA, 367-376.

Kriens, P., Hall, R. S., 2006. OSGi RFC-0112 Bundle Repository [pdf]. Available at: http://www.osgi.org/download/rfc-0112_BundleRepository.pdf

Kriens, P., Nicholson, R., Little, M., Bosschaert, D., Rellermeyer, J. S., 2011. RFP 133 Cloud Computing [pdf]. Available at: http://www.osgi.org/wiki/uploads/Design/rfp-0133-Cloud_Computing.pdf

Kruchten, P., 1995, Architectural Blueprints — The "4+1" View Model of Software Architecture. *IEEE Software*, Vol. 12, Issue 6, pp. 42-50.

Le Berre, D., Rapicault, P., 2009. Dependency management for the Eclipse ecosystem: Eclipse P2, metadata and resolution. *In: Proceedings of the 1st International Workshop on Open Components Ecosystem*. ACM.

Mell, P., Grance, T., 2011. The NIST Definition of Cloud Computing [pdf]. *National Institute of Standards and Technology*. Available at: http://public.dhe.ibm.com/common/ssi/ecm/en/wsd14071usen/WSD14071USEN.PDF

Natis, Y. V., 2011. Hype Cycle for Cloud Application Infrastructure Services (PaaS), 2011. Gartner.

Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D., 2009. The Eucalyptus Open-Source Cloud-Computing System. *In: Cluster Computing and the Grid, 2009. CCGRID '09. 9th IEEE/ACM International Symposium*, pp. 124-131.

O'Brien, T., Casey, J., Fox, B., Van Zyl, J., Moser, M., Redmond, E., Shatzer, L., 2008 Maven: The Complete Reference. O'Reilly

OSGi Alliance, 2011. OSGi Service Platform, Core Specification, Release 4, Version 4.3, available at http://www.osgi.org/Download/Release4V43

Rubio, D., 2009. Pro Spring Dynamic Modules for OSGi™ Service Platforms. Apress.

van Vliet, J., Paganelli, F., van Wel, S., Dowd, D., 2011. Elastic Beanstalk. *O'Reilly Media*.

Wolff, E., 2011. Cloud Foundry: Cloud PaaS von VMware-Open Source für Public und Private Cloud. *In: Java Magazine*, 2011.

Zahariev, A., 2009. Google App Engine. *In: Seminar on Internetworking.*