A Novel Approach for Developing Interoperable Services in Cloud Environment

Binh Minh Nguyen, Viet Tran, Ladislav Hluchy
Department of Parallel and Distributed Computing
Institute of Informatics, Slovak Academy of Sciences
Bratislava, Slovakia
{minh.ui, viet.ui, hluchy.ui}@savba.sk

***Abstract*— Cloud computing has seen a tremendous growth in the last five years. Along with the growth, many cloud models have been marketed. They deliver hardware and software as virtualization enabled services to users. Although cloud computing offers considerable advantages such as unlimited resources, manageability and lower investment costs but there are still barriers to exploit it. One of the barriers is the difficulties, which users are being faced when developing and deploying their own services into clouds. In this paper, we present a novel approach for developing interoperable services that can be deployed in different cloud infrastructures at the same time. The approach provides an instrument with emphasis on abstraction, inheritance and code reuse. Using the approach, cloud-based services are developed easily by extending existing abstractions classes provided by the instrument or other developers. The interoperability between different clouds is solved via the basic abstraction class of the instrument and all services are inherited and benefited from this advantage.**

Keywords-cloud computing; service development; asbtraction; inheritance; interoperability

I. INTRODUCTION

With the advance of cloud technologies, the trend of developing and deploying services/applications in cloud environment also has appeared. There are economic as well as technological reasons why an application should be developed and deployed on the cloud. On the economic side, cloud computing can provide significant cost savings due to the increased utilization resulting from the pooling of resources (often virtualized). Otherwise, cloud computing enables rapid delivery of IT services, which increases business efficiency. On the operational side, manageability, performance, and scalability are the typical reasons why service developers consider cloud computing. However, this trend is being faced two challenges:

- The Platform-as-a-Service (PaaS) type limits developers to concrete platforms and Application Programming Interfaces (APIs).
- The Infrastructure as a Service (IaaS) type provides too low-level service.

In principle, PaaS clouds offer environments for hosting and APIs for implementing applications. The platforms will manage the execution of these applications and provide some advanced features like automatic scaling. However, for existing or legacy applications, PaaS may require rewriting completely their codes using a certain dedicated platform, what is not feasible for developers and users as well. Furthermore, each platform can have different key property and API, what make moving the applications from a platform to another practically impossible.

Meanwhile, IaaS clouds provide resources (virtual machines, storage) as services where developers have full access to the resources and manipulate directly. For instance, using IaaS, they will log into virtual machines (VMs) and execute some commands or modify some files in order to create their own development and deployment platform on the machines.

Easily realize that, while PaaS binds developers into existing platforms, building services on IaaS will be their choice to meet specific requirements. However, the IaaS use is perceived as difficult with them, requiring advance computer skills for the installation and usage [1], [2] and [3]. As mentioned above, *service developers have to themselves prepare VM(s) and platform(s).* This process is similar to the age of assembly languages, where the programmers can modify every byte of memory and CPU registers. Such access is error-prone and potentially interferes with high-level automation and optimization.

Consequently, from the view of service developers, they need to have a solution which enables to simplify the development and deployment of services on IaaS clouds. The solution must allow developers to write, pack and deliver codes of the services for deployment on various clouds without any obstacle. The solution thus also provides developers with a single unified interface to overcome the incompatible API between the clouds.

In this paper, we present a novel approach for developing interoperable cloud services that can be deployed on multiple IaaS clouds at the same time. The foundation of the approach is a high-level Cloud Abstraction Layer (CAL) providing developers with a unified interface to support management of the entire service life cycle from development, through deployment to execution and operation over IaaS clouds. Based on CAL, the process of service development and deployment will be easier and simpler. More importantly, it overcomes the problem of the centralized VM manager and allows the interoperability between different clouds.

II. RELATED WORKS

Although some cloud standardization efforts and open API abstractions have emerged, such as Open Virtualization Format (OVF) [4]; Open Cloud Computing Interface (OCCI) [5]; SimpleCloud API [6]; jCloud [7]; ApacheLibcloud [8]; DeltaCloud [9] and many others, but they have not yet brought any solution for service development and deployment issue in cloud environment.

Typically, OVF allows reuse of a standard VM image on diverse clouds. Thereby, services can move from a cloud to others along with the image. Besides OVF, another project is OCCI that enables developers to manage resources from different clouds via a single unified API. Unfortunately, both standards still require many efforts from developers or users, who have to carry out a lot of the complex steps (as mentioned in Section 1) to develop and deploy their services into the IaaS clouds. Reason is the lack of a suitable programming model for the development and deployment. In addition, the standardizations force cloud providers to accept and support their products (e.g. standard image, API). Such scenario would face a very large impact on the vendor competitiveness, as it requires them to offer better quality services at lower prices without locking customers to rely on only their resources.

Instead of these de-facto standards, the open API abstractions (e.g. SimpleCloud API, jCloud, ApacheLibcloud, DeltaCloud and others) also have been designed and implemented in order to manage resources from clouds. The advantage of the API abstractions is independent from cloud vendors. However, like OCCI, these API also do not help developers to develop and deploy cloud-based services more easily than the traditional way. The developers still have to directly connect to the VMs, install and configure everything to prepare their own platforms. At the present, no API abstractions have provided functionalities for these tasks.

III. GENERAL APPROACH

Currently, services in IaaS clouds are usually developed and released in form of pre-defined images, which are very cumbersome, difficult to modify, and generally non-transferable between different clouds. For example: marketplace of AWS appliances [10], VMware Virtual Appliance [11], StratusLab [12] and so on. Though OVF can solve in part of the move issue but it almost cannot be applied to commercial clouds that have held the largest market share of cloud computing due to the competition reason.

Therefore, coming from the requirement of a solution that allows developers to use diverse clouds at the same time irrespective of the differences in cloud models (e.g. public, private), middleware (e.g. image, hypervisor) and incompatible APIs, this work proposes design and development of a high-level layer based on abstraction approach which would:

- Simplify and streamline the deployment and deployment of cloud services, not dissimilar to the way it is currently being handled in the cloud environments via mobile images.
- Remove vendor lock-in of cloud providers by a single unified interface.

For these purposes, **the approach releases cloud services as installation and configuration packages that will be installed on VMs** already deployed in cloud resources and containing only the base OS. Then, developers just choose a suitable OS provided by clouds and deploy the configuration packages to create their own services. Advantages of this service approach are:

- Most cloud infrastructures support images with base OS, so the services are easily transferrable. For instance, at the time of this writing, Ubuntu 12.04 LTS is the popular OS provides by most of providers (both public and private clouds).
- The base OS images provided by the infrastructures are usually kept up-to-date, so they are secure.
- VMs are always started correctly with cloud middleware. In contrast to image delivery of the existing approaches like OVF and the marketplaces presented above, which must have the acceptance and support from providers.
- Services can work on unknown infrastructures without changing codes implemented before.
- The approach allows taking full advantages of many existing applications (especially legacy ones) that already have had install/setup tools or via package managers of base OS (e.g. *apt*, *rpm* and *git* package of Debian/Ubuntu Linux).
- Allowing automatic deployment of developed services with near zero VM tuning.

One of the techniques supports perfectly for the abstraction approach is object-oriented programming (OOP). Relying on OOP, the approach offers services as modules or *objects*. A strong configuration and control interface together with a programming language for component control would allow this. Nowadays, many appliances are already provided in a package which specifies the configuration interface (Google Android and Apple iOS appliances for example), allowing developers to hide appliance details and thus optimize them in the manner that developers want. At the time, **users use cloud services exclusively via interface provided by developers**.

Since services and their functions are defined in form of OOP objects, they can be extended and customized in order to create new services based on the existing codes. In this way, **developers can reuse service codes via the inheritance mechanism without learning implementation detail of the origin**. The advantages of the code reuse are:

- Service developers do not have to use any middleware functionality directly. Thus, the codes of services are portable between different clouds.
- Developers just focus on service aspects, not the clouds. In this way, the approach reduces efforts to learn about cloud middleware.
- Allowing developers themselves to create PaaS and SaaS based on IaaS.

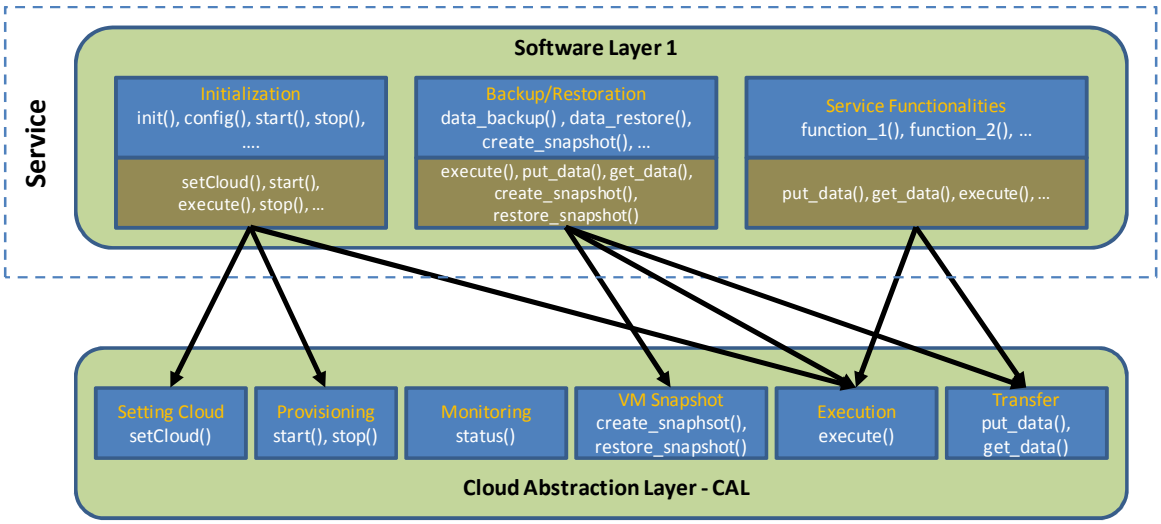

Figure 1. Inheritance feature of CAL

IV. DESIGNING

A. CAL Design

As mentioned before, CAL is designed with number of programming functions in order to manage and interact with VMs that belong to different clouds. These functions are divided into functionality groups:

- **Setting Cloud**: enables developers to set which cloud will be used. This group has only `setCloud()` function.
- **Provisioning**: consists of `start()`and `stop()` function to create and terminate the VMs.
- **Monitoring**: getting actual information of the machines (cloud provider, IP address, ID instance and so on) by `status()` function.
- **Execution**: running commands on VMs by `execute()` function.
- **Transfer** includes two functions: `put_data()` to upload and `get_data()` to download data to/from the VMs.
- **VM Snaphost**: creates/restores snapshot of VM into an image. The group involves `create_snapshot()` and `restore_snaphsot()` function.

To interact with various clouds, for each of them, we design a driver, which uses its specific API to manage VMs. Note that, the drivers do not need to implement all API functionalities that are provided by cloud vendors. Each driver only uses necessary API actions to shape CAL functions, including `start()`, `stop()`, `status()`, `create_snapshot ()` and `restore_snaphsot()`. The `setCloud()` function is call to set a driver (cloud) to use.

Otherwise, Execution and Transfer do not use any APIs because no APIs provide functionalities to carry out those operations. Thus, CAL abstracts the connection and realization process and hides implementation details by the `execute()`, `put_data()` and `get_data()` functions.

Through abstract functions above, detailed interactions between CAL and clouds are hidden. Therefore, developers can manipulate multiple clouds at the same time under the unified interface without caring about how each cloud works.

B. Inheritance Features of CAL

Developers can easily create cloud services by using CAL. Fig.1 describes this. The developers just have to inherit the existing functional abstractions of CAL for creating new service functions, which can be grouped as follows:

- **Initialization**: developers just reuse the Setting Cloud functionality of CAL to select cloud in order to deploy their services. Then they create a VM on the cloud by using functions of Provisioning. The developers can add OS commands to install software packages on newly created VM by Execution. Otherwise, they also can upload initial data, their application, configuration files into the VM by Transfer.
- **Backup/Restoration**: comprises service functions to perform two separate tasks:
  - Creating/restoring snapshot for the service together with VM. For this purpose, developers inherit VM Snapshot functionalities.
  - Creating/restoring only backup of user data, developers reuse Execution and Transfer.
- **Service Functionalities**: developers can create functions for their services by reusing and combining the existing functional abstractions of CAL. For examples, for database servers, they can add a number of functions to import database, make query, and so on. The database functions are programmed based on Execution and Transfer.

One of the most important things is that during development, developers do not need to use any specific-middleware APIs or connect directly to VM as well. They

only inherit the functions provided by CAL. The developers also can select simply the target cloud to deploy their service without having to worry about incompatible cloud systems. Meanwhile, their users (distinguish from the developer) will only use of the service via Initialization, Backup/Restoration, Service functionalities. The users would not require caring about how and where the service is developed and deployed.

C. Software Layering

A software layer, which has been created by a developer, also can be used and extended further by other developers in the same way, i.e. a developer defines *software layer 1* with new functionalities on his or her user demands. In other words, the layer 1 hides implementation details of CAL in its functionalities. Similarly, other developer can define functionalities for *software layer 2* over the layer 1 by inheriting the layer 1 functionalities. As the result, each layer is practically a platform-as-a-service by itself, because users can use the service via a clear interface provided by developers.

Since higher software layers do not use any APIs provided by the cloud middleware, if CAL correctly operates on a cloud infrastructures, any services using CAL can operate correctly on the cloud infrastructures, too. The generalized problem of the interoperability of cloud applications, therefore, can be reduced into solving the problem of the interoperability of selected software layers of CAL in our development framework. Developers can easily move applications/services (software layers) without depending on clouds: using the initialization functionality of services, developers can unite simply a software layer with others. Thus, our approach enables perfectly interoperability of cloud-based services among different clouds.

V. CASE STUDIES

A. Experimental Setup

Our current implementation of CAL prototype bases on the installations of three middleware: OpenStack [13] Essex release, Eucalyptus 2.0.3 [14] (both are compatible with Amazon EC2 API [15]) and OpenNebula 3.6 release [16] (using OCCI 0.8 API specification). The middleware are configured separately. Each of them consists of a controller node, a management network (switch) and two compute nodes. For controller nodes, each server blade is equipped processor Xeon with 16 cores (2.93 GHz), 24GB of RAM and 1TB hard drive. Meanwhile, for compute nodes, each server blade is equipped processor Xeon with 24 cores (2.93 GHz), 48GB of RAM and 2TB hard drive. Linux is installed for all servers as OS. KVM hypervisor is used for all three systems. An Ubuntu 12.04 images are created and deployed on the clouds. While OpenStack, Eucalyptus are configured with Glance [17] and Walrus [18] respectively as internal image storage services, OpenNebula uses non-shared file systems [19] with transferring image via SSH for test purpose.

By default, the cloud middleware provide different VM types that are choices of physical configuration (CPU, RAM, disk space). As its design, for convenient use, CAL abstracts them into the following families: `small`, `medium` and `large`. In which, the small type is set to default instance with minimum configuration for testing. The medium and large types otherwise provide high capacity and high performance as well.

B. Development and Deployment of Cloud-based Services

As Python language [20], the abstraction layer is represented as a class (`CAL`) which provides the basic functions of VM. For each cloud infrastructure, we respectively define separate classes: `Eucalyptus`, `OpenNebula` and `Openstack`, which are the drivers of these clouds. Each the class uses the middleware-specific API and utility mechanisms (execution, transfer functionality) for the implementation. Structurally, the functions of the `CAL` class are derived from the separate classes. If we need to extend our implementation for a new cloud infrastructure, the only thing we have to do is to create the new derived class (driver) for its specific middleware functions.

The case study carried out process of development and deployment of a concrete service using CAL. Inspired by the fact that most users need a cloud-based hosting for their applications, a webhosting was built. Purposes of the service are:

- Providing a platform for development and deployment of web applications (e.g. websites or WordPress [21] blog).
- The platform can be deployed into well-known clouds of CAL without any changes in its code.
- Limiting interaction with VMs for the service user.

The service is developed based on Apache web server (LAMP). It is an open source platform that uses Linux as OS, Apache as web server, MySQL as the relational database management systems and PHP as the OOP language. The service class is programmed as follows:

```
class webhosting(CAL):

    def setCloud(self, cloud):
        CAL.setCloud(self, cloud)

    def config(mysql_password):
        …

    def start(self, image, VM_type)
        CAL.start(self, image, VM_type)
        CAL.execute(self, 'install_LAMP_command')
```

The backup and restoration functionality also are written shortly inside the `webhosting` class:

```
    def create_snapshot(self):
        CAL.create_snapshot(self)

    def restore_snapshot(self):
        CAL.restore_snapshot(self)

    def data_backup(self):
        CAL.get_data(self, ' ')

    def data_restore(self):
        CAL.put_data(self, ' ')
```

Besides fundamental functions such as `setCloud()`, `config()`, `start()`, `stop()`, `create_snapshot()` and `restore_snapshot()`, the webhosting provides specific function for web application developers, including:

- `data_backup()` and `data_restore()` backs up/restores only user data (e.g. web pages) on the server to local.
- `upload()` uploads files or web packages into the hosting server. The function will return URL of the websites.
- `run()` runs web configuration commands on the server (e.g. changing directory name of the web package, copying files)
- `mysql_command()` runs MySQL statements (e.g. creating database, username, password or setting access privilege for a database).

```
def upload(self, dir_package):
    CAL.put_data(self, dir_package)
    CAL.execute(self, ' ')

def run(self, user_command):
    CAL.execute(self, user_command)

def mysql_command(self, MYSQL_statement):
    CAL.execute(self, MySQL_statement)
```

After development, the service can be deployed simply by running commands:

```
service = webhosting()
service.setCloud('cloud_name')
service.config('mypass')
service.start()
```

At the time, users can use the hosting service through its existing functions. There are two types of web applications:

- Flat sites are simple web pages. They are called *flat* because they do not use any database on the server.
- Dynamic sites are built based on database. Examples of this type are e-commerce websites, forums (content management system) and blogs.

The webhosting service supports both types. For flat sites, web designers just use `upload()` function to upload their site packages into the server. Then, the webhosting service will return URL for these sites. For example, a flat site with name "*mysite*" is hosted on the server by:

```
service.upload('mysite')
```

For dynamic sites, developers can program new software layers by inheriting webhosting service functions. Each the new software layer serves a specific purpose of a dynamic site. In order to demonstrate expansibility of the webhosting service as well as CAL, we continue to develop a blog WordPress service dealing with the layer of webhosting. This service is inherited all existing functions and deployed on the server provided by the webhosting. The following presents some codes of it:

```
class wordpress(webhosting):

    def config(self, database_pass):
        webhosting.mysql_command(self, ' ')

    def start(self):
        webhosting.run(self, 'install wordpress')
        …

    def stop(self):
        …
```

Blog users only have to enter password for their database on the server when creating their blogs. A blog is created simply by commands:

```
blog = wordpress()
blog.config('mysecret')
blog.start()
```

In the studies, hosting service is the software layer 1 (equivalent to PaaS – see Fig. 1). According to the user requirements, we can go further with providing many other hosting functions such as FTP transfer or phpMyAdmin [22] interface for MySQL databases. Meanwhile the WordPress service acts as software layer 2 (equivalent to Software as a Service - SaaS) over the layer 1. We also can to deploy other SaaS services (wiki pages, forum for instance) based on the webhosting. The most important thing is the codes of both software layers can be deployed on any cloud infrastructures by changing the name of target cloud in the `setCloud()` function. Therefore, CAL enables the ability to deliver services among different clouds without any obstacle. Furthermore, users of the services do not need to worry about the VM management because the process of service deployment is automatic.

C. Experimental Results

To evaluate operation of webhosting service, WordPress service and CAL, the process of service deployment is tested on three existing cloud installations with various VM types. The experimental measurement is repeated 20 times for each of the VM type of each cloud. The average values of deployment time that are summarized in Table I and Table II. The duration time is calculated in second.

TABLE I. WEBHOSTING DEPLOYMENT TIME

	VM Type	Duration
OpenNebula	small	655.718
	medium	666.898
	large	674.093
Eucalyptus	small	365.66
	medium	373.248
	large	388.702
OpenStack	small	215.7
	medium	225.132
	large	236.491

TABLE II. WORDPRESS DEPLOYMENT TIME

	VM Type	Duration
OpenNebula	small	113.014
	medium	112.128
	large	110.768
Eucalyptus	small	84.66
	medium	79.66
	large	78.652
OpenStack	small	48.495
	medium	46.215
	large	45.985

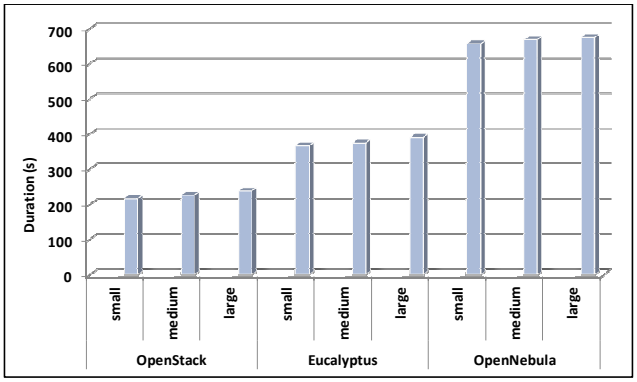

Figure 2. Deployment time of webhosting service

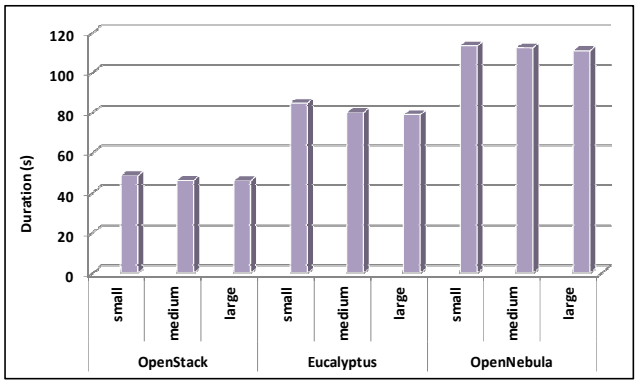

Figure 3. Deployment time of WordPress service

The results also are illustrated by diagrams in Fig. 2 and Fig. 3. There are some observations that can be made from inspecting the results. First, CAL operates well with the well-known clouds. Second, the webhosting and WordPress service can be deployed on all these clouds. Third, in the case of comparison between clouds, deploying the webhosting service on OpenStack is faster than on Eucalyptus (approx. 41%) and OpenNebula (approx. 67%). The reason is that OpenNebula installation uses non shared file system and image is transferred between nodes via SSH. Additionally, while OpenStack is kept up-to-date with consecutive versions, Eucalyptus only supports open source with an old version. This is importance factor that can explain why OpenStack achieves higher performance than Eucalyptus cloud. Four, in the same cloud, since attributes of VM types are different. Therefore, deployment of the webhosting service into medium or large type is faster than small type. However, using medium and large VM, the process of service deployment is still slower because the VM startup needs more time. Finally, deploying WordPress on webhosting server of OpenStack requires less time than Eucalyptus (approx. 43%) and OpenNebula (approx. 57%). Otherwise, for small web applications like WordPress, the disparity in deployment time is quite small when carrying out the test with diverse VM types on the same middleware.

Although the process of deploying services takes a long time. However, since the process is realized automatically, the time for deployment is always less than manipulation of the traditional approaches.

VI. CONCLUSION

In this paper, we presented the novel approach for developing interoperable cloud-based services that are treated as objects with strongly defined interfaces. The foundation of the approach is a high-level abstraction layer that provides basic functionalities of VM for each known cloud middleware. Based on the layer, process of service development and deployment is easier: developers will build their services by inheriting the existing functionalities of the abstraction layer without using any middleware APIs as well as directly connecting to the VMs. Thus, developed services are independent of infrastructures and they can be deployed on the diverse clouds. In this way, our approach enables the service interoperability, which is one of the invaluable features for cloud computing.

ACKNOWLEDGMENT

This work is supported by projects CLAN No. APVV 0809-11, VEGA No. 2/0054/12, CRISIS ITMS: 26240220060, SMART II ITMS: 26240120029.

REFERENCES

[1] Ramakrishnan, L., Jackson, K. R., Canon, S., Cholia, S., Shalf, J.: Defining Future Platform Requirements for e-Science Clouds. ACM Proceedings of the 1st ACM symposium on Cloud computing. 2009. p. 101-106

[2] Goscinski, A., Brock, M.: Toward dynamic and attribute based publication, discovery and selection for cloud computing. Elsevier, Zv. Future Generation Computer Systems. 2010. p. 947-970

[3] Curry, R.; Kiddle, C. ; Mirtchovski, A. ; Simmonds, R. ; Tingxi Tan.: A Cloud-based Interactive Application Service. IEEE proceedings of the Fifth International Conference on e-Science. 2009. P. 102-109

[4] Open Virtualization Format. http://dmtf.org/sites/default/files/OVF%20Overview%20Document_2010.pdf.

[5] Metsch, T., Edmonds, A., Nyrén, R.. 2011. Open Cloud Computing Interface – Core.. http://forge.gridforum.org/sf/go/doc16161

[6] SimpleCloud API. http://simplecloud.org

[7] jCloud. http://jclouds.org

[8] ApacheLibcloud. http://libcloud.apache.org

[9] DeltaCloud. http://deltacloud.apache.org

[10] AWS marketplace. https://aws.amazon.com/marketplace

[11] Ishtiaq Ali and Natarajan Meghanathan. Virtual machines and networks – installation, performance, study, advantages and virtualization options. International Journal of Network Security & Its Applications (IJNSA), Vol.3, No.1, 2011.

[12] StratusLab marketplace. http://stratuslab.eu/doku.php/install:marketplace

[13] OpenStack. http://openstack.org

[14] Nurmi, D. Wolski, R.; Grzegorczyk, C.; Obertelli, G.; Soman, S.; Youseff, L.; Zagorodnov, D.: The Eucalyptus Open-Source Cloud-Computing System. IEEE proceedings of the ninth IEEE/ACM International Symposium on Cluster Computing and the Grid. 2011. p. 124-131.

[15] Amazon EC2 API. http://docs.amazonwebservices.com/AWSEC2/latest/APIReference

[16] Milojičić, Dejan, Llorente, Ignacio M. ; Montero, Ruben S.: OpenNebula: A Cloud management Tool. Journal IEEE Internet Computing. 2011. vol 15. issue 2. p. 11-14

[17] Gregor von Laszewski, Javier Diaz, Fugang Wang, Geoffrey C. Fox. Comparison of Multiple Cloud Frameworks. Proceeding of IEEE Cloud 2012. P. 734 – 741.

[18] Lonea, A.M. A survey of management interfaces for eucalyptus cloud. In proceeding of 7th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), 2012. P. 261 – 266.

[19] Xiaolong Wen; Genqiang Gu; Qingchun Li; Yun Gao; Xuejie Zhang. Comparison of open-source cloud management platforms: OpenStack and OpenNebula. Proceeding of IEEE 9th International Conference on Fuzzy Systems and Knowledge Discovery. 2012. p. 2457-2461.

[20] Python programming language. http://python.org

[21] WordPress blog source. http://wordpress.org

[22] phpMyAdmin. http://phpmyadmin.net