Computing and Informatics, Vol. 32, 2013, 461–485

A GENERIC DEVELOPMENT AND DEPLOYMENT FRAMEWORK FOR CLOUD COMPUTING AND DISTRIBUTED APPLICATIONS

Binh Minh Nguyen, Viet Tran, Ladislav Hluchý

Institute of Informatics
Slovak Academy of Sciences
Dúbravská cesta 9
845 07 Bratislava, Slovakia
e-mail: {minh.ui, viet.ui, hluchy.ui}@savba.sk

Communicated by Jacek Kitowski

Abstract. Cloud computing has paved the way for advance of IT-based on demand services. This technology helps decrease capital expenditure and operation costs, solve scalability issue and many more user and provider constraints. However, development and deployment of distributed applications on cloud environment becomes a more and more complex task. Cloud users must spend a lot of time to prepare, install and configure their applications on clouds. In addition, after development and deployment, the applications almost cannot move from one cloud to another due to the lack of interoperability between them. To address these problems, in this paper we present a novel development and deployment framework for cloud distributed applications/services. Our approach is based on abstraction and object-oriented programming technique, allowing users to easily and rapidly develop and deploy their services into cloud environment. The approach also enables service migration and interoperability among the clouds.

Keywords: Cloud computing, distributed application, abstraction, object-oriented programming, interoperability

Mathematics Subject Classification 2010: 68-M14

1 INTRODUCTION

Cloud computing is described as a business model for on-demand delivery of computation power, in which consumers pay providers what they used ("pay-as-you-go"). The critical point that distinguishes cloud from other computing paradigms is that cloud itself is considered fully virtualized, making illusion of the unlimited resources. Therefore, cloud enables providers to optimise their provisioning and users gain utility computing on demand. With the trend towards cloud model, individuals and businesses have gradually changed their habits of using the computational resources from their local computers or servers to data centers managed by third party. Conceptually, cloud computing gathers key features like high availability, flexibility and elasticity, that intends to reduce total cost and decrease risk for both users and providers. Today, consumers can buy raw resources, platforms or applications over cloud infrastructures. In the language of this market, the commodities are usually referred to X-as-a-service (XaaS) paradigm (XaaS is commonly used to express "anything/something"-as-a-service), which mainly includes Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS).

In principle, cloud-based appliances or services often provide higher availability and lower cost than the traditional IT operations. For this reason, there is now a strong trend of deploying and migrating appliances to cloud. This process could be realized by using PaaS or IaaS. On the one hand, the PaaS clouds provide platform (programming language, databases and messaging) for implementing services and environments for hosting them. The platform also manages the execution of these appliances and optionally offers some advanced features like automatic scaling. Thus, this model allows developers to simply create their cloud services without the need to manually configure and deploy virtual machines (VMs). On the other hand, the IaaS clouds provide raw resources (VMs, storages) where the users have full access to the resources and manipulate them directly in order to create their own platform and deploy services there.

It can be realized easily that, while PaaS binds developers into certain existing platforms, building cloud services in IaaS will be their choice to meet specific requirements. However, the use of IaaS is perceived as difficult with the developers, requiring advanced computer skills for the installation and usage. Otherwise, since several commercial cloud systems have been already marketed, the problem of interoperability between these systems also arises, i.e. it would be possible and feasible to move appliances from a cloud provider to another, or to deploy an existing appliance on resources provided by several different cloud providers. Such possibility would have very large impact on the competitiveness of providers, as it would require them to offer better quality services at lower prices without forcing customer to rely on their resources through vendor lock-in.

Although several standardizations and solutions in this area have emerged, they have not yet brought any comprehensive solution for the service development and deployment issue on IaaS clouds. Therefore, from the view of general cloud users, they need to have an instrument, which can solve the problem. In this way, cloud service developers will achieve the highest work performance from cloud computing. For example, developers can write, pack and deliver the codes of their services for deployment on various IaaS clouds. In addition, the developers also may manage the services via a unified interface without worrying about the incompatible application programming interfaces (APIs). Thereof, cloud users are not IT experts, who can exploit cloud computing easily to perform their complex tasks.

The work presented in this document is dedicated to innovative research and development of an elastic instrument (called high-level Cloud Abstraction Layer – CAL) allowing easy development and deployment of services on resources of multiple infrastructure-as-a-service (IaaS) clouds simultaneously. A primary design idea of CAL was introduced in [1] before; relying on this, we continue to mature it with developing components and experimenting effects on cloud environment. Technically, the CAL provides a total novel approach with emphasis on abstraction, inheritance and code reuse. Then, cloud-based services can be developed easily by extending available abstractions classes provided by the CAL or other developers. Interoperability between different clouds is solved by the basic abstraction classes of the CAL and all services are inherited and benefited from the advantage. The work does not stop on theoretical work but also continues in applying CAL to build cloud services in order to deal with real problems.

2 PRESENT STATE OF THE ART AND RELATED WORKS

This section presents and points out the advantages as well as disadvantages of existing solutions that relate to the issue of service development, deployment and interoperability. These solutions could be divided into two groups according to their characteristics, namely de-facto cloud standards and abstractions APIs.

2.1 De-Facto Cloud Standards

Open Virtualization Format (OVF) [2] is one of the numerous projects of Distributed Management Task Force (DMTF) [3]. The project aims at creating a VM standard that ensures a flexible, secure, portable and efficient way to distribute VMs between different clouds. The VM standard is called virtual appliances [4]. Users can package a VM in OVF and distribute it on a hypervisor. The OVF file is an XML file that describes a VM and its configuration. Application-specific configurations can be packaged and optimized for cloud deployment, as multiple VMs, and maintained as a single entity in OVF format. For instance, an application may comprise the web server, application server and database server. Hypervisors that support OVF include VMware [5], Oracle VirtualBox [6], Citrix XenServer [7], and IBM AIX running on Power systems [8], Microsoft Hyper-V [9] and IBM Mainframe Linux z/VM [10]. Although appliances/services can move among various clouds along with the OVF image, creating cloud-based services still requires considerable effort from developers, who must carry out a lot of complicated steps such as preparing VM and development platform, deploying and managing the services. Additionally, all of the operations are performed without any utility functionalities or any supports from cloud providers. This causes time-consuming and increased cost. Otherwise, the incompatible APIs issue also brings about the service operations are not guaranteed when they migrate from one cloud to another.

Open Cloud Computing Interface (OCCI) [11] is a recommendation of the Open Grid Forum [12]. The OCCI is a RESTful Protocol [13] and API for all kinds of management tasks. OCCI was originally initiated to create a standard management API for existing IaaS clouds, allowing users to manage and use various clouds under single unified interface that intends to enable cloud interoperability. Technically, users send HTTP/HTTPS requests in turn to OCCI web service with actions and parameters to control and manage resources on cloud. The parameters are standardized and defined in a XML or JSON (JavaScript Object Notation) file, which is attached to the requests. OCCI web service then responds to the user with XML content, including necessary information of cloud resources. Although OCCI enables cloud users to manage resources from different clouds at the same time, unfortunately, the users still have to directly connect to VMs in order to create their own platform for service development and deployment. In this way, the service migration is almost impossible.

In addition, the critical weakness of the cloud standardizations is that they force cloud providers to accept and support their products. Such scenario would have a very large impact on competitiveness of the cloud vendors, as it would require them to offer better quality services at lower prices without locking customers to rely on their resources.

2.2 Abstraction APIs

Similar to OCCI, abstraction APIs (e.g. Simple Cloud API [14], Deltacloud [15], SAGA API [16]) have been created for managing resources from various clouds at the same time. These APIs support a set of functionalities that are a common denominator among different providers and hide technological differences by their interfaces. Therefore, the APIs do not require any support from the cloud vendors. A typical solution is Simple Cloud API, which provides a management interface in form of PHP library. Functionally, the PHP implementation of the Simple Cloud API uses the *Factory* and *Adapter* design patterns [17]. To work with a particular cloud, user calls the appropriate factory method with an array of configuration parameters. The class returned by the factory method is an adapter to a particular cloud service. The adapter maps the Simple Cloud API calls to the service-specific calls required by each cloud provider.

Apache Deltacloud is categorized into cloud research about provisioning and management. Deltacloud defines a REST-based API for managing and manipulating cloud resources in a manner that isolates the API client as much as possible from the particulars of specific cloud API's. The Deltacloud API is implemented via HTTP request methods. All data is communicated via the REST interface to a Deltacloud server, which similarly has a REST interface. To simplify operation of the REST interface, the Deltacloud project provides a CLI (commandline interface) tool, as well as client libraries in Ruby, Java, C, and Python. This makes Deltacloud differ from other solutions including Simple Cloud API, Apache Libcloud [18], etc. in which all these libraries are tied to a specific programming language − Simple Cloud API to PHP, Apache Libcloud to Python. Deltacloud, on the other hand, is entirely independent of languages and is the only cloud abstraction API that can also be used as a web service. The advantage of this approach is that through the use of widely accepted and existing standards such as HTTP and XML, an open architecture is created independency from platforms and programming languages.

SAGA API is an API for deploying applications to distributed computing systems. The API defines an abstraction layer for scheduling jobs to grid or cloud infrastructures, including execution, management functions of files or data. This abstraction scheduler is essentially a resource broker (which is about negotiating between those who want to consume resources and providers). Applications are thus represented as jobs. However, since it is built based on grid computing, there are drawbacks when applying it to cloud. In real time, the SAGA API applications are not services. The life cycle of the applications only consists of three stages: submit to computational resources, run or calculate, return results or outputs. After that, the used resources will be terminated. Consequently, the API is only suitable for computational applications (e.g. calculation, simulation), not for development and use of cloud services.

Besides the API abstraction tools presented above, there are others, which could be mentioned such as: jclouds [19], boto [20], Apache Cloudstack [21], enStartus enterprise [22], Right Scale [23], Scalr [24] and so on. All of them are developed based on the abstraction approach in order to provide cloud users a tool for managing resources from clouds. As identified before, the advantage of the API abstractions is independent of cloud vendors. However, like OCCI, these abstractions do not help developers develop and deploy services more easily than the traditional way. Thus, the developers still have to prepare a platform and develop the services by connecting directly to VMs. Although there are abstraction APIs, which offer support for service deployment via scheduling job mechanism like Simple Cloud API, jcloud and SAGA API, due to limitations of the mechanism, they cannot be used for developing services. At present, no APIs have provided a comprehensive solution for both development and deployment tasks.

Clearly, the problem of service development and deployment is a big gap of cloud computing today. There are three reasons for this:

- PaaS cloud type limits developers/users to concrete platforms and APIs.
- Lack of suitable programming model for service development on IaaS.
- Lack of interoperability between different IaaS clouds.

So far, there are not any ideas or solutions, which address and resolve this difficult problem. In comparison with all solutions above, our approach has differences and advantages in the following aspects:

Support for both development and deployment of cloud services – CAL allows developers to write, package and deploy services into IaaS clouds. The approach customization and extension are emphasized: developers can take service developed by others and customize/extend to new capabilities without access to the original code.

Enabling service interoperability between cloud infrastructures – CAL is independent from underlying cloud middlewares, so developers can choose the target cloud infrastructure to deploy the developed services and manage them in a unified interface.

Value-added providers – the approach enables value-added providers, who do not require supporting CAL, to simultaneously achieve easy use for their users.

3 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 cloud providers. For example, marketplace of AWS appliances [25], VMware Virtual Appliance [26] StratusLab [27], etc. provide a large number of services in that way. Though OVF can partially solve image move problem, it almost cannot be applied to commercial clouds that have held the largest market share of cloud computing uses (see Section 2.1).

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

- simplify and streamline the deployment of cloud services, not dissimilar to the way it is currently being handled in the cloud environments via standard VM images;
- remove vendor lock-in of cloud providers by a single unified interface.

For this purpose, **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 operation system (OS). Then, users just choose an OS provided by clouds (e.g. Ubuntu 12.04 LTS) and deploy the configuration packages to create their own services. The advantages of this service approach are as follows:

- 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 provided by most of providers (both public and private clouds).

- VMs are always started correctly with cloud middlewares, unlike image delivery of the OVF approach and the marketplaces presented above, which must have 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).
- Allowing automatic deployment of developed services with near zero VM tuning.

One of the techniques supporting the abstraction approach is **object-oriented programming** (OOP). The reason is that the *encapsulation* and *polymorphism* mechanism of OOP allow hiding state details and standardizing different interaction types, and those are the beginnings of the abstraction [28, 29]. Through these mechanisms, OOP brings the ability of defining a common interface over different implementations and behaviors of various systems. In addition, the OOP *inheritance* mechanism also allows creating new abstraction layers over an initial abstraction layer easily [30].

Relying on OOP, we offer 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 (Android and iOS appliances are typical examples) allowing developers to hide appliance details and thus optimize them in the manner the developers want. At the same 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 details of the origin**. To achieve this feature, CAL provides basic functionalities of VMs with base OS. Developers will inherit the code and modify/add functions related to their services. CAL then acts as foundation for development and deployment of cloud services on IaaS clouds. The advantages of the code reuse are as follows:

- Service developers do not have to use any middleware functionality directly. As a consequence, the codes of services are portable between different clouds.
- Developers just focus on service aspects, not on the clouds; in this way, the approach reduces the efforts to learn about cloud middlewares.
- Enabling developers themselves to create PaaS, SaaS based on IaaS.

4 DESIGN OF HIGH-LEVEL CLOUD ABSTRACTION LAYER

The high-level abstraction layer for IaaS cloud aims to be a ubiquitous environment serving distributed application and services. Therefore, the layer is expected to be flexible enough to support several different clouds. The term "development" means that the CAL allows developers to create cloud services easily and independently from underlying infrastructures (create one, run anywhere). "Deployment" means that the layer allows the developers to deploy the developed services easily to chosen infrastructure via a simple, unified interface. In order to achieve this purpose, as presented in Section 3, the approach of this research is based on abstraction technique.

4.1 Abstraction of Cloud Resources

4.1.1 Data Abstraction

Theoretically, an abstraction hides details of an entity. In the context of using cloud computing, there are many "entities" that need to be mentioned. However, due to abstraction approach and OOP, the entities are encapsulated as data abstractions, which can be used easily via programming methods. Figure 1 shows the hierarchical data abstraction of cloud resources.

The cloud resource abstraction is divided into two parts, namely VM and middlewares. While the abstraction of VM allows simplifying the use with VM, the abstraction of middlewares enables CAL users (service developers) to manage VM simply under a single interface.

4.1.2 VM Abstraction

The abstraction of VM comprises two components: hardware and software. CAL represents them as data to be used without understanding implementation details.

Hardware Abstraction: Like traditional computer, hardware of VM includes devices provided as *"virtual"*, consisting of CPU, memory, disk space, OS platform and network. As a rule, IaaS clouds offer their users VM types, in which each type implies a set of the hardware devices. In other words, these types are abstractions of VM hardwares. For example, Amazon EC2 enables users to select several types of VM with different attributes [31]. The *Small Instance* type is equipped with one virtual core CPU, 1.7GB of memory, 160GB of disk storage and 32-bit or 64-bit platform of OS. The other types are equipped with more powerful hardware and bring higher performance for VM. Besides, most of public clouds provide the network by associating automatically a public IP address with VM. For open source clouds, there are some exceptions (e.g. OpenStack [32] which requires users to associate floating IP address manually).

In the heterogeneous environment of multiple clouds, the existing VM types usually are diverse. Therefore, to use these clouds simultaneously, CAL defines

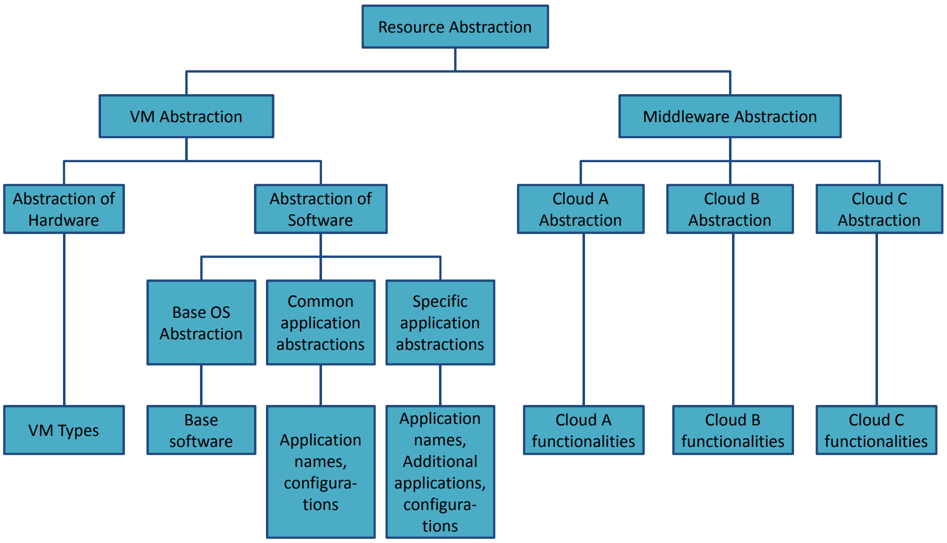


Fig. 1. High-level abstraction of cloud resources

a high-level abstraction of VM hardwares based on the types provided by the clouds. More specifically, all types of VM are grouped into the following classes: `small`, `medium` and `large`, which are abstract data and can be used within the OOP methods. The `small` is abstraction of the smallest VM types of well-known clouds. Similarly, the `medium` is the abstraction of one of the medium types and the `large` is the abstraction of the largest type. CAL also ensures that created VMs always have access via IP address provided by cloud middlewares. Most clouds support this. For the exceptions presented above, CAL provides a mechanism that enables automatic association between VMs and floating IP addresses. This mechanism is implemented based on API cloud functionalities provided by cloud vendors. Furthermore, CAL allows capability of extending to support more other types according to user requirements.

Due to hardware abstraction, CAL users just select one of the types above to create a VM with different configurations. The `small` is set to default type, which is suitable for test purposes. So, users can deploy services on VM without setting this type in functions of CAL. On the other hand, users set other types when they want to use higher configurations.

Software Abstraction: Software of VM is the collection of programs that provides instructions for controlling the machine what to do and how to do it. In the context of virtualization, the software refers to base OS, common and specific applications. In the same way of hardware abstraction, the implementation of CAL also abstracts VM software in form of data.

- Base OS(s): VM base OS(s) are often preconfigured and released together with images. Ubuntu/Debian, RedHat Linux, CentOS, Fedora, Windows Server, etc. are the popular base OSs provided and used in clouds. The abstraction of base OS(s) is described by abstract data, containing the OS name and its versions (e.g. `Ubuntu 12.04`, `CentOS 6.3`). The data can be used easily in OOP methods. Thereby, CAL enables users to achieve the simplification in creating VM with a desired OS.
- Common applications: Common applications of VM are applications that are used widely. In most cases, they can be installed with default configurations via advance packaging or manager tools (e.g *apt* package manager of Ubuntu and Debian). MySQL, TomCat, Apache, Java and Python libraries, etc. are typical examples of the common applications. There is some information, which is required to abstract the applications covering their names and configurations: *application name* is a word or term used for identifying the application. The name thus discriminates applications from each others. Additionally, the application names normally are expressed together with development versions, which are used to assign in increasing order and correspond to new release in the application. For instance, the data abstraction of database application is defined as "`MySQL 5.1`", in which "`MySQL`" is the application name and "`5.1`" is its version. *Application configurations* are initial settings for an application. Application configuring allows users to choose suitable parameters, ensuring the application work as they want. Username and password setting are often required for the configuration process.
- Specific applications: Specific applications are constituted from many components that can be common applications or file configurations. In general, these applications comprise a core and additional supplements. The process of abstracting these applications is carried out one after another as follows: installation of the core, installation of the supplements and configuration of whole applications. For example, to build a web server (specific application), developers have to install and configure Apache, MySQL and PHP (common applications) in turn. Then they configure parameters of the whole server such as privacy, web directory, etc.

4.1.3 Middleware Abstraction

One of the objectives of CAL is to manage resources from many cloud middlewares at the same time. Currently, most of the clouds provide APIs, which are efficient management means for users. Many existing cloud tools have been built based on the APIs (see Section 2). Essentially, API is an abstraction of cloud functionalities such as creation, termination, VM snapshot and so on. As mentioned before, API has many different forms and each cloud usually offers a separate API that differs from others. To achieve the objective defined in the previous section, CAL implements the APIs as data abstractions. In this way, each cloud has a data abstraction for its own and the abstraction provides only necessary functionalities to manage resources of the cloud. At the higher level, abstraction of middlewares encapsulates the API data abstractions in the form of new type, allowing manageability of multiple middlewares under a single unified interface.

Abstraction of middlewares can be considered as a common denominator of functionalities provided by various clouds. Basically, prevalent cloud functionalities include: *creation*, *termination*, *VM description*, *snapshot creation/restoration*. They are basic management functions for developing cloud services on VM. Most of them are supported by every cloud through their APIs. Beside the basic functions presented, there are many other functionalities such as IP allocation and association, user key creation, deletion, etc. According to requirements of CAL users, the implementation of API data abstractions can be extended in order to add more functions for the abstraction of middlewares.

4.2 CAL Architecture

Architecture of CAL is depicted in Figure 2. It contains three components: interface, drivers and data repository. The interface provides interaction between CAL and its users. Therefore, from the view of service developers, it is visible. Otherwise, the drivers and data repository are designed to hide under the interface. Since the number of variations of VMs is often limited (some major base OS flavors or VM types), making these functionalities operate correctly on a given cloud infrastructure can be solved with reasonable efforts.

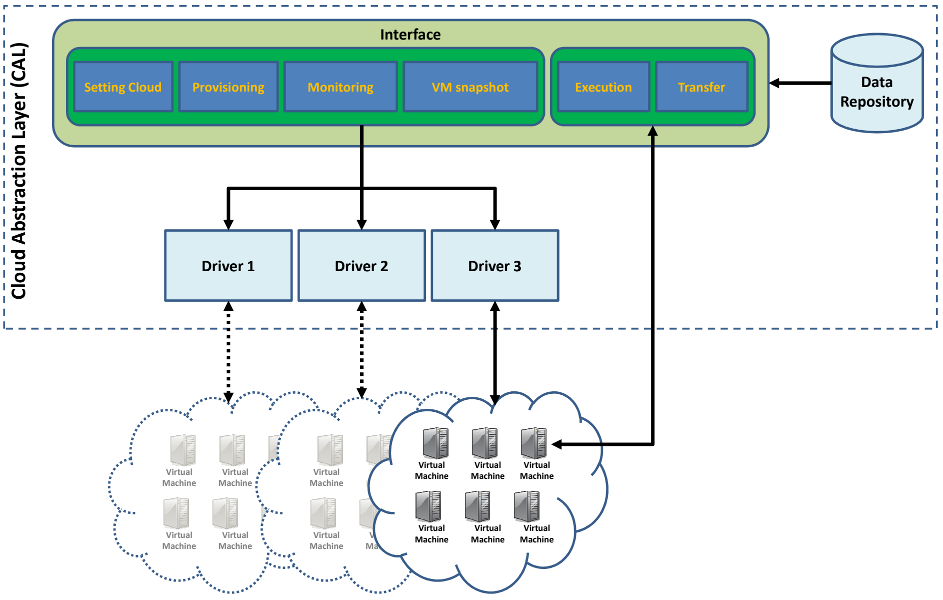


Fig. 2. CAL architecture

4.2.1 CAL Interface

The interface is designed to provide functional abstractions of the cloud resources. Using the CAL, service developers just reuse the functions of CAL interface to create their services. For easier understanding, the functions are divided into numerous groups, consisting of:

Setting Cloud: enables developers to set which cloud will be used. This group has only the `setCloud()` function.

Provisioning: consists of the `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 the `status()` function.

Execution: running commands on VMs by the `execute()` function.

Transfer includes two functions: `put_data()` to upload and `get_data()` to download data from the VMs.

VM Snaphost: creates/restores snapshot of VM into an image. The group involves the `create_snapshot()` and `restore_snaphsot()` function.

While the functions of Provisioning, Monitoring and VM Snapshot group use cloud APIs to implement actions of VM, Execution and Transfer do not. The reason is that there are no APIs that support the operations. The advantage of CAL is to provide developers with functionalities in order to overcome the restrictions. In this way, CAL abstracts the connection, realization process and hides implementation details by the functions of Execution and Transfer. As the result, developers can run commands or upload, download their data without having to directly connect to VM. Since VMs are normally connected via public IP addresses under authentications (e.g. key pair, usernames, passwords), Execution and Transfer functionalities are used for all VMs even when they belong to different clouds.

4.2.2 CAL Drivers

CAL may have many drivers. Each driver is a module of *API data abstraction*, which allows CAL to manage resources from a cloud via its API. As mentioned before, the number of drivers is in proportion to the number of clouds that a CAL supports. For example, if a CAL supports Amazon EC2 and OpenNebula cloud, it means that it will have two drivers. When a CAL needs to expand its support for other clouds, the only thing to do is creating the new driver without changing the original codes, including interface and existing drivers. The relationship between each driver and the interface within a CAL is similar to the relationship of a hardware device (e.g. graphic card) and an application or OS in traditional computer, where hardware driver acts as translator between the device and application. Indeed, the drivers of CAL also act as intermediaries to translate the developers commands (via functionalities of CAL interface) to clouds. Because of the incompatibility of APIs, every cloud must have a driver of its own.

Because CAL provides only basic functionalities of VM, the drivers thus do not need to implement all API actions. The CAL functionalities are implemented through APIs covering Provisioning, Monitoring and VM snapshot.

4.2.3 CAL Data Repository

Beside interface and drivers, data repository plays the part of memorizing parameters related to VM and services during CAL usage. For example, the repository stores *name* of default base OS(s), VM snapshot, VM IP address, VM ID and other configuration parameters (e.g. administrator name, password, database name). A data repository for CAL brings many advantages for service developers, including:

- Optimization of the CAL usage – the developers do not have to repeat the values of VM IP address, VM ID or other parameters (e.g. service password) in every functional abstraction. All of them are stored automatically in the data repository, thus simplifying CAL use.
- Elimination of repeating service installations, since VM snapshot names are stored in the repository. Service developers can optimize the VM start process by checking existing services. If a service is available, its snapshot will be used to create VM. Otherwise, the developers will deploy developed services in the normal way (with a base OS).

4.3 Inheritance of CAL Functions

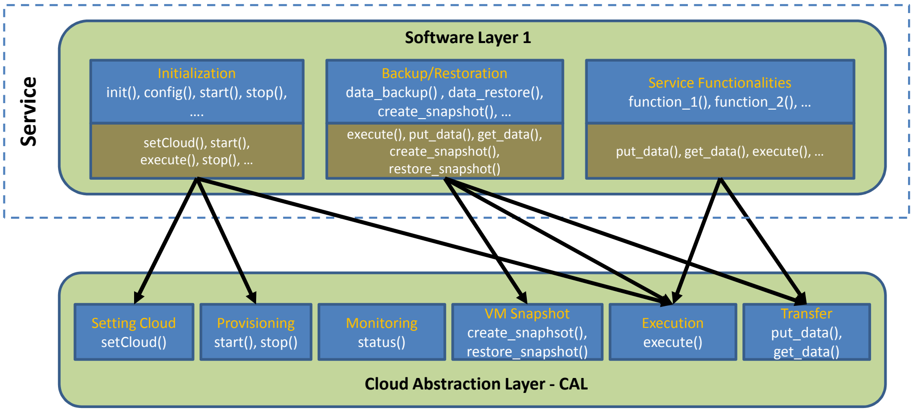

Service
Software Layer 1
Initialization
init(), config(), start(), stop(), ...
setCloud(), start(), execute(), stop(), ...
Backup/Restoration
data_backup() , data_restore(), create_snapshot(), ...
execute(), put_data(), get_data(), create_snapshot(), restore_snapshot()
Service Functionalities
function_1(), function_2(), ...
put_data(), get_data(), execute(), ...
Setting Cloud
setCloud()
Provisioning
start(), stop()
Monitoring
status()
VM Snapshot
create_snaphsot(), restore_snapshot()
Execution
execute()
Transfer
put_data(), get_data()
Cloud Abstraction Layer - CAL

Fig. 3. Inheritance feature of CAL

As mentioned before, developers can easily create cloud services by using CAL. This is shown in Figure 3. 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 the functions of Provisioning. The developers can add OS commands to install software packages on newly created VM by Execution. Otherwise, they also can upload their initial data or applications into the VM by Transfer.

Backup/Restoration: comprise service functions to perform two tasks:

- *Creating/restoring VM snapshot* for the service. For this purpose, the developers inherit VM Snapshot functionalities.
- *Creating/restoring user backup data.* The developers reuse Execution and Transfer.

Service functionalities: Developers can create other functions for their services by reusing and combining the existing functional abstractions of CAL. For example, 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, the developers do not need to use any specific-middleware APIs or connect directly to VMs as well. They only inherit the functions provided by CAL. The developers can thus simply select the target cloud to deploy their service without having to worry about incompatible cloud systems. Meanwhile, their user (distinguish from the developer) will just use the service via Initialization, Backup/Restoration and service functionalities. The user would not have to care about how and where the service is developed and deployed.

4.4 Software Layering

A software layer, which was created by a developer, can also be used and further extended by other developers in the same way. Figure 4 shows the inheritances by multiple service developers. In this figure, the first developer defines software layer 1 with new functionalities on demand from his or her users. In other words, software layer 1 hides implementation details of CAL in its functionalities. Similarly, the second developer defines functionalities for software layer 2 over the layer 1 by inheriting layer 1 functionalities. As the result, each software layer is practically a platform-as-a-service by itself, because users only use the services via a clear interface provided by the developers without interacting directly with VMs on clouds. For concrete example, a developer creates a LAMP stack (Linux-Apache-PHP-MySQL) layer that is equivalent to web-hosting platforms for his or her service users. Another developer can use this LAMP layer to provide web applications (e.g. websites, wiki pages and forum) without manipulating the cloud infrastructures.

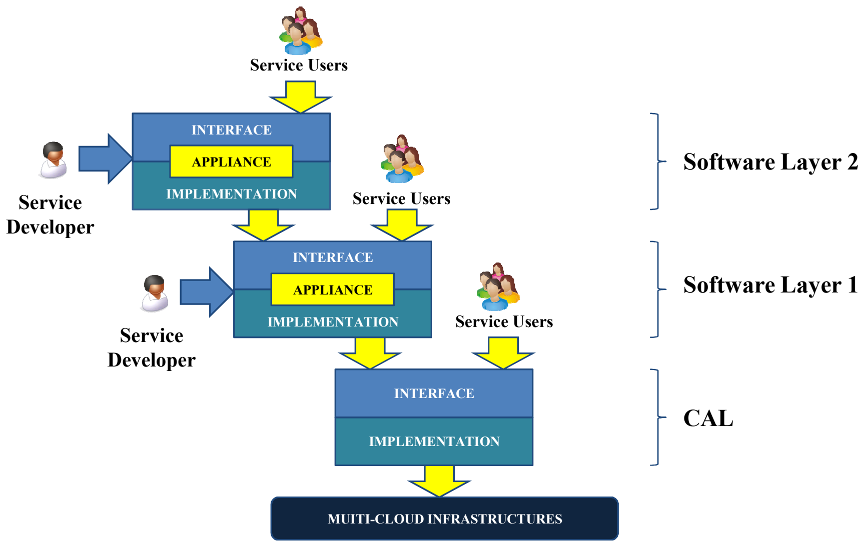


Fig. 4. Software layering of CAL development framework

Since higher software layers are independent from cloud infrastructures, if CAL operates with multi-clouds correctly, any services using CAL will also operate correctly on the infrastructures. Generalized interoperability problem of cloud systems can be reduced into the interoperability problem of selected software layers in this development framework. Developers can easily move services (software layers) among various clouds without depending on providers. Consequently, the approach enables perfectly interoperability of cloud services.

5 CASE STUDIES

5.1 Experimental Setup

Our current implementation of CAL prototype is based on the installations of three middlewares: OpenStack Folsom release, Eucalyptus 2.0.3 (open source version) [33] and OpenNebula 3.6 [34]. The purpose of this setup is to provide VMs that belong to those middlewares at the same time. Tests are successful if and only if all VMs have SSH access. For that purpose, all three middlewares are installed and configured separately in the data center of Institute of Informatics, Slovak Academy of Sciences (II – SAS) [35]. Each of them consists of a controller node, a management network (switch) and at least two compute nodes. For controller nodes, each server blade is equipped with Xeon processor including 16 cores (2.93 GHz), 24 GB of RAM and 1 TB hard drive, meanwhile for compute nodes, each server blade is equipped with Xeon processor with 24 cores (2.93 GHz), 48 GB of RAM and 2 TB hard drive. Linux is installed for all servers as OS. KVM hypervisor is used for all three systems.

Ubuntu 12.04 images are created and deployed on the clouds. While OpenStack and Eucalyptus are configured with Glance [36] and Walrus [37] respectively as internal image storage services, OpenNebula uses non-shared file systems [38] with transferring image via SSH for test purpose.

5.2 Development and Deployment of Cloud Monitoring Service

The realistic services are the best way to show and demonstrate the CAL effects. In the direction, this section presents the development and deployment of a cloud monitoring service for distributed systems based on CAL. Additionally, the layering feature of the services is also demonstrated here.

A monitoring service must have the ability to provide data usage of components within VM and fundamental aspects of appliances in that VM. As a consequence, the service needs to be adaptable and extensible in order to support the expanding functionality. To address all of the requirements and functionalities, the main features for service to be taken into account are as follows:

- Scalability: ensures that that the monitoring can cope with large numbers of VMs.
- Adaptability and extensibility: ensure that the monitoring framework can adapt to varying computational load.
- Federation: ensures that any VM which resides on various clouds can be monitored.

To establish such features, the monitoring service is built based on Nagios core framework application. First reason of the choice is that Nagios core is a powerful monitoring solution that is used by large IT organizations [39]. Otherwise, it also has been one of the most prevalent monitoring solutions known in open source community. The second reason is that the core does not contain any checking tools (called Nagios add-ons) at all. Due to this important feature, Nagios core can provide a robust, reliable and extensible framework for any type of check that a user can come up with. Currently, the most general Nagios add-ons are:

- NRPE (Nagios Remote Plugin Executor) provides the ability to monitor VM metrics (e.g. disk space, CPU workload).
- Nagios BPI (Bussiness Process Inteligence) creates a way to visualize business process health by grouping hosts and services together, and creating rules to discern the true health of the network infrastructure as it relates to the business.
- Nagiosgraph extracts information from the Nagios output, processes it, and then insert it into one or more round-robin database (RRD) files. The plugin also embedded RRD file directly into Nagios interface in form of graphs as trend reports.

5.2.1 Nagios Monitoring Framework Service

Using Python language [40], the high-level abstraction layer is represented as "`CAL`" class, which provides the basic functions of VM. For each cloud infrastructure, we define separate classes: `Openstack`, `Eucalyptus` and `OpenNebula`, which are the drivers of these clouds.

Figure 6 describes the software layering of monitoring framework service. In the context of the implementation using CAL, the framework can be considered as platform-as-a-service layer that can be used for many different monitoring purposes. Based on the layer, developers can program to provide various specific monitoring services by installing Nagios add-ons. In this way, the Nagios add-on services are equivalent as software-as-a-service layer over the monitoring framework.

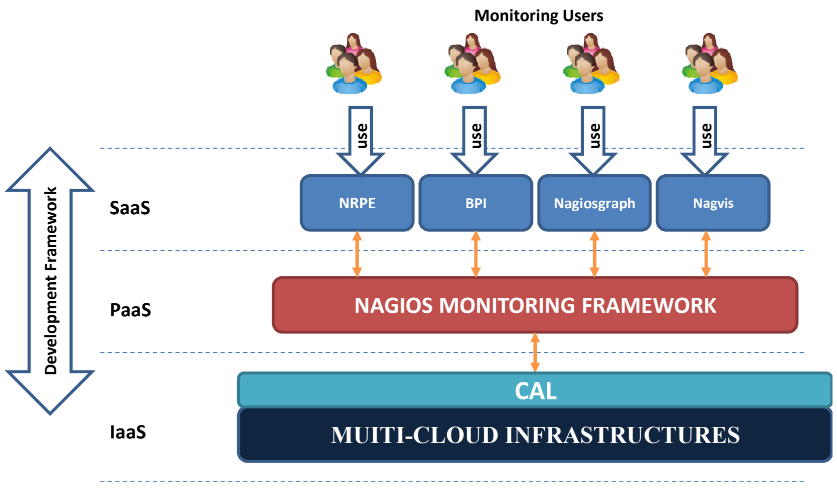


Fig. 5. Nagios monitoring framework service

Although Nagios core is a large and complicated application that normally requires expert computer skills to install and configure, using CAL, the development process of monitoring framework service can be carried out quite easily. The monitoring framework service is implemented as `Nagios` class. Service methods are created by reusing (inheritance mechanism) the CAL methods. Beside common methods, namely `setCloud()`, `start()`, `stop()`, `create_snapshot()` and `restore_snapshot()`, the service provides two specific methods for SaaS layer developers:

- `put_data()` uploads plugin packages or configuration file to the server.
- `execute()` runs Nagios configuration commands inside server.

The monitoring framework service is implemented to operate on Linux server. Installation commands of Nagios are predefined in configuration files, which are uploaded to VM. Then the files are called to perform via CAL `execute()` method. The service codes are developed as follows:

```
class Nagios(CAL):
#inheriting all CAL functions

    def setCloud(self, cloud):
        #choosing cloud to deploy service
        CAL.setCloud(self, cloud)

    def config(admin_name, password):
        #configuring service admin name and password
        ...

    def start(self, baseOS, instanceType):
        #start service
        CAL.start(self, baseOS, instanceType)
        CAL.put_data(self,'config_file')
        CAL.execute(self, 'run_config_file')
        ...

    def stop(self):
        CAL.stop(self, ' ')

    def put_data(self, data_file):
        CAL.put_data(self, 'data_file')

    def execute(self, command):
        CAL.execute(self, 'command')

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

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

Due to CAL, the service can be deployed on all three cloud middlewares with diverse VM types. This process is realized by the following commands:

```
service = Nagios()
service.setCloud('OpenStack')
service.config('admin', 'mysecret')
service.start()
```

After `start()` method is called and performed, it returns web interface URL of the Nagios framework service. Users can login by admin name and password that they just have set.

5.2.2 NRPE Monitoring Service

As mentioned before, NRPE is one of the most popular add-ons of Nagios core. This section describes the implementation and use of NRPE service, which can monitor resources and services running on cloud VMs. In the development framework, NRPE monitoring service is equivalent to SaaS layer because it is developed based on a platform (Nagios monitoring framework service). NRPE add-on provides a lot of different monitoring functionalities such as check disk, CPU load, memory, users, number of running/zombie process, service status (http, apache, MySQL), and so on. All of the functions are defined in the object definition files. In this way, users have the ability to choose which resource or services are monitored.

The NRPE functions are implemented as methods of `NRPE` class. Similar to monitoring framework service, the installation commands also are predefined in configuration files that are uploaded to VM to execute. Otherwise, to remove or stop NRPE service on monitoring framework, the removing file with un-installation commands defined inside is used.

```
class NRPE(Nagios):
#inherit Nagios functions

    def start(self):
        Nagios.put_data(self, ‘config_nrpe_file’)
        Nagios.execute(self, ‘run_config_nrpe_file’)

    def stop(self):
        Nagios.put_data(self,‘remove_config_nrpe’)
        Nagios.execute(self, ‘run_remove_config_nrpe’)
        ...

    def add_host(self, IP_instance):
        #config for Nagios server
        Nagios.put_data(self, ‘add_server_file’)
        Nagios.execute(self, ‘run_add_server_file’)

        #config for host
        Nagios.put_data(self,‘IP’,‘config_host_file’)
        Nagios.execute(self,‘IP’,‘run_config_host’)

    def remove_host(self, IP_instance):
        ...
```

Transparently, during the service development, only monitoring framework service functions are used. Developers do not need to know about VM, SSH, SCP commands as well as implementation details of the monitoring framework service. The NRPE monitoring is called through the following commands:

```
check = NRPE()
check.start()
check.add_host('VM_IP_address')
```

To remove a monitored host from the server, users only enter the command:

```
check.remove_host('VM_IP_address')
```

5.3 Experimental Results

To evaluate operation of CAL, Nagios framework monitoring service and NRPE service, the deployment process is tested on three cloud installations with various VM types. The experimental measurement is repeated 20 times for each of the VM type of each cloud. The NRPE service deployment process is experimented based on Nagios framework layer, which already has been deployed into the VMs of three cloud middlewares. The average times are summarized in Table 1. The duration time is calculated in seconds.

	VM Type	**Nagios framework monitoring service**	**NRPE service**
OpenStack	*small*	346.296	20.276
	medium	341.407	19.128
	large	334.121	17.765
Eucalyptus	*small*	626.204	58.421
	medium	621.812	53.357
	large	618.698	52.245
OpenNebula	*small*	808.742	70.603
	medium	796.186	66.679
	large	788.233	60.603

Table 1. Deployment time results of Nagios framework monitoring service

Figure 6 represents the deployment time results of Nagios framework monitoring service described in Table 1. There are some observations that can be made from inspecting the results: first, Nagios framework monitoring service can operate well on all three cloud infrastructures at the same time without changing its implementation codes. Second, in comparison between OpenStack with the rest, the middleware gains better performance. This is demonstrated by the experiment results presented in Table 1. Specifically, the deployment process of Nagios core framework service using OpenStack is faster than using Eucalyptus (by approx. 49 %) and using OpenNebula (by approx. 57 %).

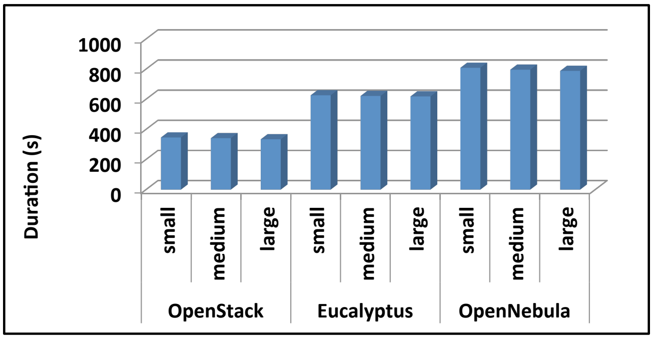


Fig. 6. Deployment time of Nagios framework monitoring service

The deployment time results of NRPE service are illustrated by the diagram in Figure 7. The experiment proves that NRPE service operates well on Nagios monitoring framework deployed on the different clouds. More importantly, the development and deployment process of NRPE service emphasize that CAL software layering feature as well as the feasibility of our approach.

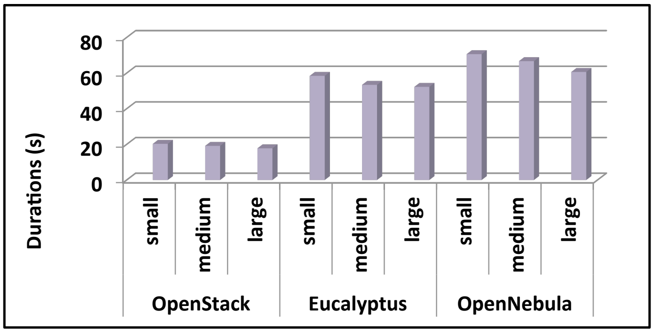


Fig. 7. Deployment time of NRPE monitoring service

In summary, the case studies give the following important outcomes:

- CAL operates well with the installed cloud middlewares. It provides the development and deployment framework for service developers and users.
- The framework enables software layering, in which each layer is the foundation for new layers over it. During the development, deployment and use process, they only inherit and reuse available functions of preceding layers instead of directly using any functionality of VM and cloud middlewares.
- The developed services were tested and work well.

- The developed services are independent of cloud infrastructures and they have the ability to be deployed on different clouds at the same time.
- 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.

6 CONCLUSIONS

The study described in this paper was carried out in the emerging context of cloud computing with the issue linked to the development and deployment of services over IaaS clouds at once. In this direction, this research focused on building a tool that enables every service to be able to develop and deploy into different clouds without requiring additional complicated work form developers and special supports from providers or middlewares. The tool was named "CAL" that brings the following results:

Mechanism to simply develop and deploy services in different IaaS clouds: cloud services are treated as objects with strongly defined interfaces. The services thus are developed based on abstraction approach and OOP. CAL acts as development framework for the services. It includes basic functionalities of VMs that belong to many clouds. With the available functions, developers easily define the interface for their services without the necessity to access application code directly; in other words, without connecting directly to VMs to configure, install the applications. In addition, developers also can customize, extend services developed by others to create new services without access to original codes. Due to CAL, the services are developed and used with minimum VM manipulations. The mechanism thus allows simplification in development and deployment of services within cloud environments.

Mechanism to enable interoperability of cloud computing services: service codes that are implemented using CAL can be packed and delivered for deployment on various clouds without obstacles such as incompatible APIs, virtualization technologies or other middleware components. This means, the codes are written once but can be used on many clouds without re-implementation. In this way, this mechanism allows achieving the interoperability, which is one of the invaluable features for cloud computing today.

Adding values for cloud providers: due to the abstraction approach of CAL, cloud services are developed and deployed independently of providers. So, the providers do not have to change in order to support the services. This still ensures the competitiveness among the vendors. Otherwise, they achieve benefits when service developers/users exploit cloud resources more easily, exciting the growth of consumer market for IaaS cloud providers.

Since CAL services are independent from underlying infrastructures, they can be published in a marketplace that allows other service developers or pure users to download and use them without coding. Consequently, in the near future, we will continue to build the marketplace for CAL services.

Acknowledgements

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

REFERENCES

[1] Nguyen, B. M.—Tran, V.—Hluchý, L.: Abstraction Layer for Development and Deployment of Cloud Services. In Computer Science, Vol. 13, 2012, No. 3.

[2] Open Virtualization Format web site. Available on: `http://www.dmtf.org/standards/ovf`, 2013.

[3] Distributed Management Task Force web site. Available on: `http://dmtf.org`, 2013.

[4] Kecskemeti, G.—Terstyanszky, G.—Kacsuk, P.—Nemeth, Z.: An Approach for Virtual Appliance Distribution for Service Deployment. Future Generation Computer Systems, Vol. 27, 2011, No. 3, p. 280–289.

[5] Li, P.: Selecting and Using Virtualization Solutions: Our Experiences with VMware and VirtualBox. Journal of Computing Sciences in Colleges, Vol. 25, 2010, No. 3, pp. 11–17.

[6] Watson, J.: VirtualBox: Bits and Bytes Masquerading as Machines. Linux Journal, Vol. 2008, No. 166.

[7] Takemura, C.—Crawford, L. S.: Citrix XenServer for the Enterprise. In The Book of Xen: A Practical Guide for the System Administrator, William Pollock, San Francisco 2009, p. 159.

[8] Tzortzatos, E.—Bartik, J.—Sutton, P.: IBM System z10 Support for Large Pages. IBM Journal of Research and Development, Vol. 53, 2009, No. 1, pp. 17:1–17:8.

[9] Leinenbach, D.—Santen, T.: Verifying the Microsoft Hyper-V Hypervisor with VCC. 2nd World Congress of Formal Methods, Eindhoven 2009.

[10] Schwarz, E. M.—Kapernick, J. S.—Cowlishaw, M. F.: Decimal Floating-Point Support on the IBM System z10 Processor. IBM Journal of Research and Development, Vol. 53, 2009, No. 1, pp. 4:1–4:10.

[11] Open Cloud Computing Interface web site. Available on: `http://occi-wg.org`, 2013.

[12] Open Grid Form web site. Available on: `http://www.gridforum.org`, 2013.

[13] Edmonds, A.—Metsch, T.—Papaspyrou, A.—Richardson, A.: Toward an Open Cloud Standard. Internet Computing, Vol. 16, 2012, No. 4, pp. 15–25.

[14] Simple Cloud API web site. Available on: `http://www.simplecloud.org`, 2013.

[15] Apache Deltacloud web site. Available on: `http://deltacloud.apache.org`, 2013.

[16] Kawai, Y.—Iwai, G.—Sasaki, T.—Watase, Y.: SAGA-Based File Access Application over Multi-filesystem Middleware. In Proceeding of 19th International Symposium on High Performance Distributed Computing, Chicago 2010, pp. 622–626.

[17] Amies, A.—Sluiman, H.—Tong, Q. G.—Liu, G. N.: Developing and Hosting Applications on the Cloud. IBM Press 2012.

[18] Apache Libcloud web site. Available on: `http://libcloud.apache.org`, 2013.

[19] jclouds web site. Available on: `http://www.jclouds.org`, 2013.

[20] Klaver, A.: Python in the Cloud. Linux Journal, Vol. 2011, No. 7, p. 210.

[21] Baun, C.—Kunze, M.—Nimis, J.—Tai, S.: Apache Cloud-stack. Cloud Computing, Springer, Berlin–Heidelberg 2011, pp. 49–62.

[22] enStartus enterprise. Available on: `http://www.enstratus.com`, 2013.

[23] Mikkilineni, R.—Sarathy, V.: Cloud Computing and the Lessons from the Past. International Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises, Groningen 2009, pp. 57–62.

[24] Moreno-Vozmediano, R.—Montero, R. S.—Llorente, I. M.: Elastic Management of Cluster-Based Services in the Cloud. 1st Workshop on Automated Control for Datacenters and Clouds, Barcelona 2009, pp. 19–24.

[25] Amazon Web Service Marketplace web site. Available on: `https://aws.amazon.com/marketplace`, 2013.

[26] VMware Virtual Appliance Marketplace web site. Available on: `http://www.vmware.com/appliances`, 2013.

[27] StratusLab Marketplace web site. Available on: `http://stratuslab.eu`, 2013.

[28] Page-Jones, M.: Encapsulation and Connascence. Fundamentals of Object-Oriented Design in UML, Indianapolis, Addison-Wesley 1999, pp. 210–212.

[29] Akin, E.: Object-Oriented Programming Concept. Object-Oriented Programming via Fortran 90/95, Cambridge University Press 2003, p. 37.

[30] Budd, T. A.: Inheritance and Substitution. An Introduction to Object-Oriented Programming (3rd Edition), Corvallis, Oregon, Addison-Wesley 2001, p. 161.

[31] Amazon Elastic Compute Cloud web site. Available on: `http://aws.amazon.com/ec2`, 2013.

[32] Pepple, K.: Deploying OpenStack. O'Reilly Media 2011.

[33] Nurmi, D.—Wolski, R.—Grzegorczyk, C.—Obertelli, G.—Soman, S.—Youseff, L.—Zagorodnov, D.: The Eucalyptus Open-Source Cloud-Computing System. 9th IEEE/ACM International Symposium on Cluster Computing and the Grid (CC-GRID '09), pp. 124–131.

[34] Dejan, M.—Llorente, I. M.—Montero, R. S.: OpenNebula: A Cloud Management Tool. Internet Computing, IEEE 15, 2011, No. 2, pp. 11–14.

[35] Institute of Informatics, Slovak Academy of Sciences web site. Available on: `http://www.ui.sav.sk`, 2013.

[36] von Laszewski, G.—Diaz, J.—Wang, F.—Fox, G. C.: Comparison of Multiple Cloud Frameworks. Proceedings of IEEE Cloud 2012, pp. 734–741.

[37] Lonea, A. M.: A Survey of Management Interfaces for Eucalyptus Cloud. Pproceedings of 7th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI) 2012, pp. 261–266.

[38] Wen, X.—Gu, G.—Li, Q.—Gao, Y.—Zhang, X.: Comparison of Open-Source Cloud Management Platforms: OpenStack and OpenNebula. Proceeding of 9th

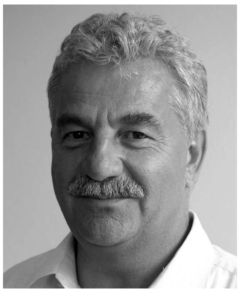
IEEE International Conference on Fuzzy Systems and Knowledge Discovery 2012, pp. 2457–2461.

[39] Ma, M.—Kouril, D.—Prochazka, M.—L'Orphelin, C.—Lequeux, O.—Veyre, P.—Triantafyllidis, C.—Kanellopoulos, C.—Korosoglou, P.: EGI Security Monitoring. Proceedings of The International Symposium on Grids and Clouds (ICGC 2012), Taipei 2012.

[40] Python programming language web site. Available on: `http://www.python.org`, 2013.

Binh Minh Nguyen received his M. Sc. in informatics from Tambov State Technical University (Russia) in 2008. Currently, he is working as a project assistant at Department of Parallel and Distributed Computing, Institute of Informatics, Slovak Academy of Sciences (Slovakia). He is also final year Ph. D. student at Faculty of Informatics and Information Technology, Slovak University of Technology in Bratislava. He is the author and co-author of several scientific papers. His research interests include cloud computing, distributed systems, service-oriented architecture and data integration.

Viet Tran is a researcher at Institute of Informatics, Slovak Academy of Sciences (Slovakia) from 1996. He received his Ph. D. degree in 2002 in parallel and distributed computing. He participated in a number of 4th, 5th, 6th and 7th EU FP projects as team leader and work-package leader. He is the scientific coordinator of several Slovak projects (VEGA, APVV). His research topics are focused on distributed computing and cloud computing. He is also the author and (co-)author of scientific books and over 100 scientific papers. He is a member of program committees and reviewer of many international scientific conferences and workshops.



Ladislav Hluchý is the Director of the Institute of Informatics of the Slovak Academy of Sciences and also the Head of the Department of Parallel and Distributed Computing at the institute. He received M. Sc. and Ph.D. degrees, both in computer science. He is project manager and work-package leader in a number of 4th, 5th, 6th and 7th EU FP projects, as well as in Slovak projects (VEGA, APVV). He is a member of the IEEE. He is also the (co-)author of scientific books and numerous scientific papers, contributions and invited lectures at international scientific conferences and workshops.