Experience in Practical Implementation of Abstraction Interface for Integrated Cloud Resource Management on Multi-Clouds

Huioon Kim, Hyounggyu Kim, Kyungwon Chun and Youngjoo Chung
School of Electrical Engineering and Computer Science, GIST
123 Cheomdangwagi-ro, Buk-gu, Gwangju 61005 – Republic of Korea
[e-mail: pcandme@gist.ac.kr, khg@gist.ac.kr, ruddyscent@gmail.com, ychung@gist.ac.kr]
*Corresponding author: Youngjoo Chung

Received September 20, 2016; revised November 26, 2016; accepted November 30, 2016; published January 31, 2017

Abstract

Infrastructure-as-a-Service (IaaS) clouds provide infrastructure as a pool of virtual resources, and the public IaaS clouds, e.g. Amazon Web Service (AWS) and private IaaS cloud toolkits, e.g. OpenStack, CloudStack, etc. provide their own application programming interfaces (APIs) for managing the cloud resources they offer. The heterogeneity of the APIs, however, makes it difficult to access and use the multiple cloud services concurrently and collectively. In this paper, we explore previous efforts to solve this problem and present our own implementation of an integrated cloud API, which can make it possible to access and use multiple clouds collectively in a uniform way. The implemented API provides a RESTful access and hides underlying cloud infrastructures from users or applications. We show the implementation details of the integrated API and performance evaluation of it comparing the proprietary APIs based on our cloud testbed. From the evaluation results, we could conclude that the overhead imposed by our interface is negligibly small and can be successfully used for multi-cloud access.

Keywords: Cloud computing, multi-cloud, cloud resource management interface

https://doi.org/10.3837/tiis.2017.01.002 **ISSN : 1976-7277**

1. Introduction

It is within bounds to say that we are now living in the *Cloud Computing* era. Since the term and concept of cloud computing first appeared, its popularity has been growing rapidly not only in the IT world but also in business and scientific areas during decade [1]–[5]. Giants in the IT industry such as Amazon [6], Google [7], Apple [8] and Microsoft [9] already started the cloud computing business and have successfully deployed their own cloud services. As a result, *infrastructure* such as CPU and storage, *platform* and even *software* can be provided to users *as a service* on a pay-per-use basis over the Internet [10]–[12].

Among the typical categories by the offerings, Infrastructure as a Service (IaaS)-style clouds provide users with exclusive infrastructure that usually consists of computing resources such as virtual machine (VM) instances, storage and network. Users build and deploy their own services or applications on the resources and easily scale up and down the virtual infrastructure by adding more resources and paying only for the additions [13]. Most of the tasks in managing a cloud, therefore, reduce to the management of the resource lifecycle. This resource lifecycle management is done through a certain software stack of each type of resource, whose interfaces are usually provided as the form of application programming interfaces (APIs) by the cloud providers or third-party developers.

As the number of clouds available per user grows and as issues of using a single cloud only have been introduced, it has come to the fore to utilize multiple clouds concurrently and collectively. This new type of usage of cloud computing called *Inter-Cloud Computing* has been proposed and explored by pioneering researchers [14]–[17]. Users who rely solely on a single cloud are at risks ranging from resource unavailability to vendor lock-in. The use of the inter-clouds can help to remove or at least mitigate these risks. One of the inter-cloud types is *Multi-Cloud* where multiple clouds are used independently by a user or a service, which is the primary topic of this paper. More details about the inter-cloud and multi-cloud will be described in the next section.

One of the issues of the use of multiple clouds is heterogeneity of the interface. Different clouds provide different APIs for accessing and managing the virtual infrastructure they offer, which makes it difficult to develop an application for deploying across multiple clouds. Developers should learn how to use different APIs when they switch from one cloud to another, which would contribute to the increase of the development cost. In order to address this interoperability issues in using multi-clouds, there have been many efforts and research including cloud interface standards [18]–[20], multi-cloud abstraction libraries [21]–[23] and integrated management of clouds [13], [24]–[32]. In this paper, we explore the previous approaches and present our own solution, an integrated interface for multi-clouds with emphasis on the resource lifecycle management that plays a role as an abstraction API for the underlying multi-clouds. This abstraction interface provides a uniform and integrated way for accessing and using resources over the multiple clouds, playing a role as a single entry point. We also show the implementation of the integrated interface and evaluation of its performance compared with the interfaces dedicated to each cloud in our testbed.

This paper is organized as follows: Section 2 reviews the previous efforts for the cloud interoperability, Section 3 gives background information on Inter-cloud and Multi-cloud, and Section 4 shows our motivations for this work. The implementation details of our idea are

presented in Section 5, and the evaluation results are shown in Section 6. Finally, the conclusions and the directions of future work are given in Section 7.

2. Related Work

2.1 Cloud Interface Standards

Not long after the cloud computing emerged, a movement toward standard for cloud interface has initiated such as Open Cloud Computing Interface (OCCI) [18], Cloud Infrastructure Management Interface (CIMI) [19], Topology and Orchestration Specification for Cloud Applications (TOSCA) [20] and so forth.

The OCCI is an open standard for cloud interface providing a specification that defines a high level of RESTful protocol and API, which was proposed by the OCCI working group in the Open Grid Forum. It is a flexible API covering all kinds of management jobs that has high interoperability and extensibility leading generic implementations such as rOCCI (Ruby) [33], pySSF (Python) [34] and erocci (erlang) [35] and many adoptions by cloud providers such as OpenStack [36], CloudStack [37] and OpenNebula [13].

CIMI is an alternative of OCCI, which is a logical model and RESTful HTTP-based protocol specification for the management of resources within IaaS domain, defined and published by Distributed Management Task Force (DMTF) Cloud Management working group. The main purpose of CIMI is to provide lifecycle management of cloud infrastructure such as creation, deletion, information retrieval and alteration of the cloud resources. The representative implementation is Deltacloud [21], which will be described in the following subsection.

TOSCA, produced by TOSCA Technical Committee in Organization for the Advancement of Structured Information Standards (OASIS), has a little bit different aspects from the standards mentioned above. Its aim is to improve the portability of cloud applications and services, and the goal is facilitated by providing a language to describe a topology of cloud applications and services and relationships and operations of them independently of the service providers. One benefit of this is interoperable management of cloud resources from different service providers.

2.2 Multi-cloud abstraction libraries

There are also some implementations for a middle layer that abstracts underlying cloud resources and services and provides a uniform API for the management of the resources from multiple cloud services. The examples of these multi-cloud abstraction libraries include Deltacloud [21], jclouds [22] and LibCloud [23].

Deltacloud consists of the API server and drivers for different cloud providers. It supports three kinds of APIs: Deltacloud REST API, DMTF CIMI REST API, and AWS (EC2 and S3) API. These APIs work as a wrapper for a large number of clouds with drivers for each cloud that interact with the native cloud APIs.

The jclouds and LibCloud are programming libraries that provide portable abstraction APIs, which can be accessed using Java or Clojure (jclouds) and Python (LibCloud). Both abstract differences among the interface libraries of multiple clouds and have a large number of supporting cloud services, but unlike Deltacloud, they do not provide a development environment for a new cloud service [24].

2.3 Integrated Management of Clouds

Lots of research and efforts have been made for the integrated management and interoperability of clouds. Here, we describe some of them which investigated integrated management and provider-independent use of clouds, architecture for interoperable clouds, standardization efforts for cloud interoperability.

Harmer et at. [25] developed an abstraction layer to provide a common resource API for computing resource providers enabling cloud-provider neutral applications to be developed. The abstraction layer provides a consistent and simplified resource usage model so that users can go through the same steps, *find*, *instantiate*, *mange*, *discard*, to access and use a multitude of providers without having to know about the unnecessary details of the providers in use.

Surveys of integrated management of IaaS resources were performed and a cloud driver was developed based on Deltacloud [21] to build an interoperable solution that incorporates OpenNebula, OpenStack and Parallels Automation for Cloud Infrastructure (PACI) and provides a web dashboard and a Representational State Transfer (REST) interface library in [24]. They did performance evaluation of the exchanged data payload and time response and showed the integrated management of IaaS clouds was possible with negligible overhead avoiding the vendor lock-in problem.

The authors in [26] tried to integrate two EU-funded cloud frameworks, RESERVOIR [38] and SLA@SOI [39], using a cloud standard, OCCI. This work shows that cloud frameworks with different architectures can interoperate and how a standard approach can be used for the purpose.

Loutas et at. [27] proposed an architecture for semantically interoperable clouds, Reference Architecture for Semantically Interoperable Clouds (RASIC), with focuses on resolving the existing semantic interoperability issues and introducing a user-centric way for applications built on and deployed in clouds. In order to do that, they combined three computing paradigms, cloud computing, Service Oriented Architecture (SOA) and lightweight semantics. Furthermore, a common cloud API model was also proposed to reduce the switching costs between different clouds and remove the vendor lock-in problem together with RASIC.

Another architectural approach is presented in [28], in which the authors proposed a three-step model: *discovery*, *match-making* and *authentication* and an architectural solution based on the Cross-Cloud Federation Manager (CCFM) complying with the three-step model. Overview of the possible practical implementation of the three agents and enabling technologies were also presented.

The three-step model mentioned above was applied to the establishment of a federation using the decentralized and dynamic brokerage approach in [29]. Difficulties and challenges of the decentralized approach were also identified focusing on the federation establishment.

An overview of IaaS cloud architecture and a concept of cloud federation were described in [13]. The core components of an IaaS cloud were identified and the cloud federation architectures were classified based on the level of coupling.

Cloud interoperability and portability issues are described and categorized according to various levels: application, platform, storage, management and configuration in [30]. The common cloud infrastructure tasks are also identified and classified.

The authors in [31] presented the interoperability issues between heterogeneous IaaS clouds and emphasized the importance of standards for enabling interoperable IaaS clouds. They also described several cloud and non-cloud standards that can address the open issues effectively by combined use of them.

Heilig et al. [32] propose an algorithm to address the cloud resource management problem in multi-cloud environments that is an optimization problem for reducing the cost and time of user applications using multiple IaaS clouds. They present experimental results based on real cloud market resources and show that their approach is good enough compared to the existing approaches.

3. Use of Multiple Cloud Services

Since the concept of cloud computing was introduced, it has rapidly spread over both of industrial and academic worlds, and a number of clouds have been built and are in use independently and separately by the entities belonging to those worlds. Ability to access a multiple available clouds and limited capability of a single cloud has brought researchers new challenges and possibilities: concurrent and collective use of multiple clouds. For this reason, the term, *Inter-cloud*, was coined and started to gain popularity with the cloud people not long after the advent of cloud computing. One aspect of how an inter-cloud is constructed and used is *Multi-cloud*. More details are given in the following paragraphs.

Definition of Inter-cloud. Briefly, an inter-cloud is an interconnected *cloud of clouds* [40]. The inter-cloud computing is interconnecting multiple clouds over a local private network or public network, i.e. Internet. More formal definition of the inter-cloud computing is presented in [15], [17]:

> *A cloud model that, for the purpose of guaranteeing service quality, such as the performance and availability of each service, allows on-demand reassignment of resources and transfer of workload through a [sic] interworking of cloud systems of different cloud providers based on coordination of each consumer's requirements for service quality with each providers SLA and use of standard interfaces.*

This definition solely offers that the inter-cloud computing collectively uses multiple clouds to overcome the limitations of a single cloud use and does not specify who is responsible for it—that is, the principal operating body between the cloud providers or the clients.

Types of Inter-cloud. Inter-clouds shown in the previous literature and implementations mostly fall into the following two categories by the principal operating body [17]:

- *Federation clouds*: Inter-clouds where a set of clouds forming the federation voluntarily interconnect to and share their infrastructures with each other. Cloud providers are the main agents in this type of inter-clouds, and their will does matter to establish a federation.
- *Multi-clouds*: Inter-clouds where a set of clouds are used independently by a user or a service. The cloud providers here do not play a key role—no voluntary interconnecting to and sharing infrastructures with each other unlike the federation clouds, and users or services are directly responsible for the usage and management of each cloud involved in the aggregation.

Benefits and Possibilities. Basically, the benefits of the inter-cloud computing come from utilizing multiple clouds concurrently. Relying solely on a single cloud can introduce various

administrative and functional issues across users and providers. A single cloud has limited resources and once all the resources are exhausted, the users cannot be provided with the resources any longer until the capacity is recovered. Inter-cloud computing can address this resource limitation problem by using other clouds' resources such as computing power, storage, and other types of infrastructures. It can also give geographical location flexibility. A cloud can suffer from a region-wide unavailability problem such as power outage, and this directly prejudices the customers who rely on the cloud located in that region. There may be other regional issues across the borders of countries, cities, and so forth, for example, legislative requirements of applications on a cloud. Not every cloud provider can establish a cloud in every country or administrative region, and therefore this type of problems is hard to solve with a single cloud/provider and can be effectively mitigated with inter-cloud computing where multiple clouds in different regions are interconnected to each other. Another issue and possibility is the vendor lock-in problem. By using multiple clouds, the vendor lock-in problem can be easily avoided—in other words, the cloud users can move their workloads from one cloud to another provided by a different provider (vendor) with ease in case the cloud provider in use changes a policy or pricing and that change can create negative impact on the users and their applications.

4. Motivations

As mentioned above, the need for concurrent and collective use of multiple clouds is on the increase. There is, however, not a high possibility that customers can do that out of the box. Not all cloud providers comply with the open standards for their cloud interface and no open standard has been widely accepted nor hold a dominant position over the others. In the early stage of cloud computing emergence, the need for and importance of the standard way to access and use multiple clouds were not stressed, and nowadays the popular public cloud providers do not actively adopt the open standards partially because of competition—in other words, they want to *lock* their customers *in* their services, so-called vendor lock-in problem, with their proprietary and incompatible interfaces. This is, however, getting considered as a negative feature by the cloud users. Open-source private cloud toolkits tend to support the market leader's interface such as AWS API or not so commonly adopt a popular open standard such as OCCI, but mostly have their own interface incompatible with others as their primary interface, too.

Table 1 shows the existing resource management APIs of several popular public and private clouds by two categories: Web APIs and language-specific APIs. Language-specific APIs are divided into two groups, first- and third-party, according to the main agent of the development. Note that the multi-cloud abstraction libraries described in the previous section are commonly included in the list of the third-party APIs for each cloud, and the exceptions are OpenNebula (jclouds does not support) and Nimbus (Deltacloud and jclouds do not support). Note also that if a cloud provider provides the AWS-compatible API layer, then most of language specific APIs compatible with AWS can be used for the provider to the extent that it supports.

Our work was greatly inspired by the interface compatibility solutions for uniform management of multi-cloud resources identified in **Table 1**. The uniform access and management is the first step of the cloud interoperability [31]. A uniform interface provides a single and consistent way to manage the cloud resources on multiple clouds so that the users do not need to learn a new API when they switch from one cloud to the other. As shown in **Table 1**, AWS-compatible API, OCCI and multi-cloud libraries can cover most of the cloud

providers listed. If we use the OCCI API, for example, we can access and manage cloud resources on OpenStack, CloudStack and OpenNebula in the same manner, without the need to change the API. Also, once we build an application or service based on the OCCI API, the application or service can be deployed on any cloud that supports the OCCI API.

Table 1. Cloud resource management APIs

	Web APIs	***Language-Specific APIs***	
		First party	**Third party**
AWS	Query API (EC2), REST API (S3)	Java, PHP, Python, Ruby, .Net, Node.js, Go	Deltacloud (Ruby), jclouds (Java), LibCloud (Python), RightScale AWS (Ruby), boto (Python), typical (Java)
OpenStack	OpenStack API (REST), OCCI API, AWS-compatible API (EC2, S3)	Python	jclouds (Java), PHP OpenCloud (PHP), LibCloud (Python), Deltacloud, Fog (Ruby), .Net, pkgcloud (Node.js)
CloudStack	CloudStack API (REST), OCCI API (rOCCI), AWS-compatible API (EC2, S3)	-	Deltacloud (Ruby), jclouds (Java), LibCloud (Python)
OpenNebula	XML-RPC, OneFlow, OCCI API, AW-compatible API (EC2)	Ruby, Java (OpenNebula Cloud API)	Deltacloud (Ruby), LibCloud (Python)
Nimbus	WSRF-based API, AWS-compatible API (EC2)	-	LibCloud (Python)
Eucalyptus	AWS-compatible API	-	Deltacloud (Ruby), jclouds (Java), LibCloud (Python), RightScale AWS (Ruby), boto (Python), typical (Java)

The existing solutions, however, place more emphasis on the uniform accessibility and usability than on the integrated use of multi-clouds. We argue that in order to realize the integrated use of multi-clouds the users or applications need to be able to view the cloud resources from multi-clouds as a large and single pool of resources. The above-mentioned cloud interfaces usually treat the resources separately, i.e. as each individual pool of a cloud provider. For example, the driver for a specific provider must be described before being used in the case of Deltacloud, and the management operations perform only on the resources provided through the driver. In the case of OCCI, the entry point for each provider to be used must be specified, which means the caller knows which cloud will be used. The users or applications do not need to know the underlying cloud infrastructures—in other words, the cloud infrastructures are hidden from the viewpoint of the interface. This is called *cloud-agnostic*. We believe it is one of the key enabling concepts for the integrated use of multi-cloud. With a cloud-agnostic interface, users or applications have the capability to launch management operations on the cloud resources without knowing where they belong to. This kind of interface is one step forward from the interfaces that only provide uniformity. This is one reason why we developed our interface in this work.

The other reason is to make it fit for our purpose. In our previous work [41], we proposed a workflow-based interface for the VM lifecycle management on clouds, which provides an integrated interface that is easy-to-use and still has enough flexibility, alleviating the problems found on the existing typical cloud interfaces: APIs, command-line tools and Web UIs. The abstraction layer in the work supports only EC2-compatible clouds and management of the VM lifecycle, and therefore we felt the necessity for an interface for non-EC2-compatible clouds and management of other types of resources. We could choose one of the aforementioned multi-cloud abstraction libraries or just adopt one of the well-known standards, but we decided to developed our own because 1) the cloud-agnostic feature is needed for the integrated use of multi-cloud as mentioned above, and 2) an enough level of simplicity is required to be invoked by workflow tasks. To summarize, the result of this work will replace the abstraction layer of the system we proposed previously, thereby making it easy-to-use and flexible resource management interface for multi-clouds based on the multi-cloud abstraction layer and workflow management system, which is our future work.

5. Implementation

5.1 Cloud Testbed

In order to develop the integrated API and test it, we built two different private clouds: OpenStack and CloudStack, as well as made use of AWS as a public cloud testbed. OpenStack is one of the most popular open-source private IaaS-cloud toolkits that has a highly modular architecture, and each resource type is backed up by separate projects and the corresponding communities. It has a fast-growing community and many non-profit and for-profit organizations as members, and the aim of it is to play a role of operating system (OS) in the cloud ecosystem. CloudStack is another example of the private IaaS-cloud toolkit which is supported by Apache Software Foundation, and therefore distributed under the Apache license. It has a modular architecture and built-in high availability (HA) for the resource hosts and VMs. AWS is the dominant commercial public cloud service in the cloud world and provides full-featured cloud services including complicated and combined services as well as the basic services such as compute, storage and network. Our testbed configuration is shown in **Fig. 1**.

As shown in **Fig. 1**, the OpenStack cloud consists of two physical machines: head node and compute node and was built with Ubuntu 12.04 and OpenStack Havana (v2013.2.3). The CloudStack cloud has four physical machines: one head node and three compute nodes and built with CentOS 6.0 and CloudStack v4.3.2. The two private clouds are connected by internal network, and the public cloud, AWS, is connected with them by external network over the Internet. The RESTful API server is based on Thin web server [42] and Sinatra web application library [43] and plays a role of the entry point of the integrated API.

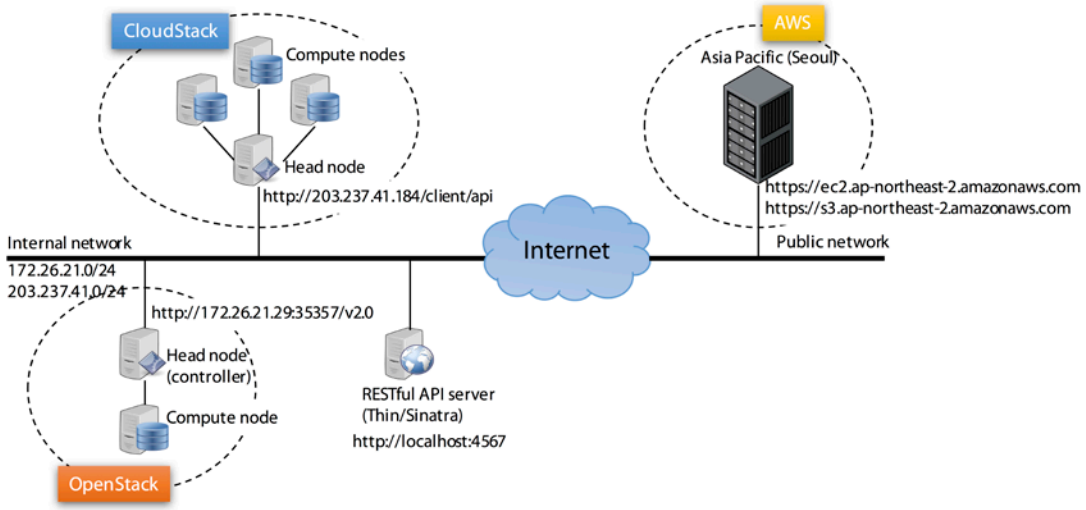


Fig. 1. Testbed overview

5.2 Design and Implementation Considerations

As mentioned in Section 4, there are two main design principals for the implementation of our integrated API: *uniformity* and *cloud agnosticism*. In this subsection, we describe some considerations about the implementation with regard to the design principals.

In the context of cloud API, uniformity means that a single set of methods is provided for all of the underlying clouds, and one method for a specific task is used in the same manner. In order to achieve this, we first identified and classified common types of the cloud resources.

The types of resources or services provided by clouds vary, but there are essential types of resources: *Compute*, *Storage* and *Network*. The compute resource is usually provided as a form of VM instance, which is a core resource of a computing system. The storage resource can be divided into two types: block storage and object storage. The block storage is provided as a form of a disk volume, and the object storage acts like an online file storage we can upload and download files. The network resource service usually provides virtual networking to isolate a network from another network or to group a bunch of VM instances to form a virtual cluster, etc. Corresponding to these basic resource types, there are three essential cloud management tasks: compute management, storage management and network management. Compute management includes listing *sizes* (hardware profiles) and VM *images* and managing VM *instance*'s lifecycle. Storage management includes managing *volumes* (block storage) and *containers*/*objects* (object storage). Network management is managing *virtual networks* and *subnets*. Details of each management task are shown in **Table 2**, and one of our implementation goals was to cover all of the tasks in the table.

The common terms of the cloud resources identified above are mostly denoted differently among the clouds. For example, the size of a VM instance is denoted as *type* (AWS), *flavor* (OpenStack) and *service offering* (CloudStack), respectively. The differences of the terms between the cloud providers are given with service or project names in **Table 3**. This inconsistency must be resolved in order to provide the uniform access and management of the target cloud resources.

Table 2. Essential cloud management tasks

Management tasks	***Details***
Compute management	• List sizes (hardware profiles) • List images • List/create/reboot/terminate VM instances
Storage management	• List/create/attach/detach/delete volumes • List/create/delete containers • List/upload/download/delete objects
Network management	• List/create/delete virtual networks • List/create/delete subnets

Table 3. Terminology comparisons

	AWS	***OpenStack***	***CloudStack***
Size (hardware profile)	Type (Elastic Compute Cloud, EC2)	Flavor (Nova)	Service offering
Image	Amazon Machine Image (AMI)	Image (Glance)	Template
VM instance	Instance (Elastic Compute Cloud, EC2)	Server (Nova)	Virtual machine
Block storage (volume)	Volume (Elastic Block Storage, EBS)	Volume (Cinder)	Volume
Object storage (container/object)	Bucket/Object (Simple Storage Service, S3)	Container/Object (Swift)	-
Network (vnet/subnet)	Virtual Private Cloud/Subnet	Network/Subnet (Neutron)	Virtual Private Cloud/Virtual Network

A cloud-agnostic interface means that the underlying cloud services are hidden from the interface so that users and applications do not need to know which cloud service will provide the requested resources, and therefore they can be separated from concerns about the underlying clouds. This puts together all cloud resources from underlying cloud infrastructures to form a large and single pool of the resources virtually, which is one of the key enabling technologies for integrated use of multi-cloud computing as mentioned above.

In order to implement the cloud-agnostic feature, firstly it needs to streamline the method parameters and make them identical between different clouds and remove what is related to a specific cloud. It also needs ID translation between low-level cloud-specific APIs and the integrated API, namely conversion between local IDs and global IDs for target resources.

Global IDs are used to distinguish cloud resources one another globally and uniquely identify one of them on the interface level, and the local IDs are not seen by users and applications. Last but not least, how to select a cloud for a specific resource is also of importance when more than two clouds are available for the desired resource. This is not an issue for the collective operations like listing, or operations that have targets specified like deleting because such operations are performed on all cloud services or a subset of clouds selected by given conditions. In the case of the operation like creating, however, it is often required that a cloud should be selected among any available clouds. This selection strategy can be various from the random choice and round-robin to the intelligent choice based on dynamic information on cloud such as capacity, utilization, etc. For simplicity, the random choice was used in this work.

5.3 Implementation Details

Our implementation was written in the Ruby programming language based on Sinatra web application library in combination with dedicated cloud APIs to access each of the clouds in our testbed. Sinatra is an open-source web application framework that provides a domain-specific language (DSL) to build web applications or services with which a RESTful API can also be developed. We made use of dedicated cloud APIs for each cloud in our testbed, which include OpenStack RESTful API [44], CloudStack Ruby client [45] and AWS Ruby SDK [46]. The overview of the implementation is shown in **Fig. 2**.

There are wrappers that handle the dedicated APIs directly and contain cloud-specific stuffs such as connection, authentication, and so on. The wrappers get user and target cloud information including credentials and entry point for the target cloud from the integrated API and try to establish a connection with the target cloud and authenticate the users. The methods included in each wrapper directly call the corresponding methods of the dedicated API, and therefore they must use the cloud-specific form of the method call. For that reason, they need the local ID of a certain resource of the target cloud instead of global ID. The ID Mapper holds the local ID – global ID map for conversion, and the methods in the wrappers query a local ID with a global ID as needed. The ID Map is saved as a file for persistency.

The Integrated API has a set of methods corresponding to each of the cloud management tasks shown in the previous subsection. It makes a method call with a global ID and options forwarded from the REST API, collects information on a user and the clouds registered to the user from *user catalog*, and supplies the information to the wrappers. It holds a list of registered clouds and select all or a subset of them by a certain selection algorithm as needed. It also uses the ID Mapper to get a name of a cloud when needed, but the name is not directly exposed to the user in any case.

The RESTful API is made with Sinatra DSL. It forwards user's input to the integrated API via a web request and gets the output as eXtensible Markup Language (XML) format by the help of XML Builder to make a response body for the web request. Its structure and examples are given in a following subsection in more detail.

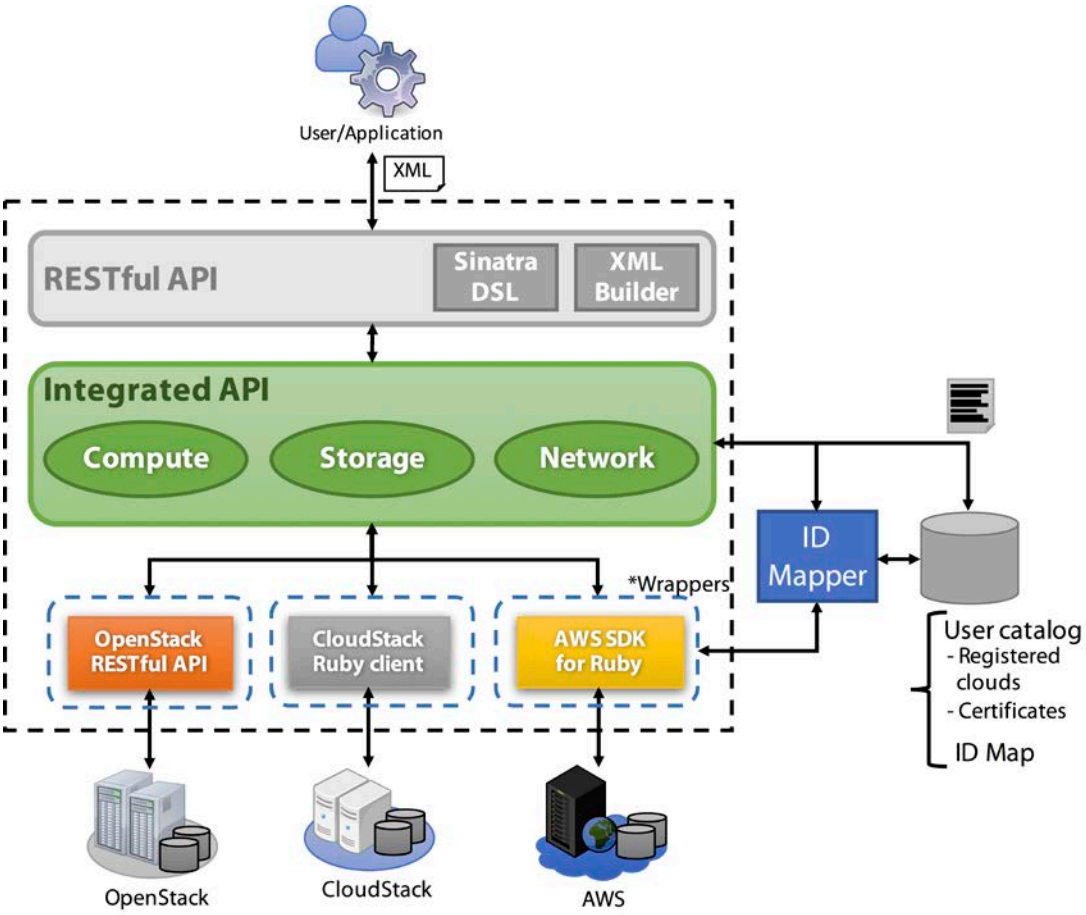


Fig. 2. Implementation overview

5.4 RESTful API

The REST stands for Representational State Transfer, which was first introduced in [47], which defines the architectural styles and design of a software architecture based on the Internet. Since its introduction, it has been widely adopted by web developers to build lightweight, scalable and reliable web services during the last decade. A RESTful API is the API built based on the REST architectural styles and design. Many cloud interface standards, multi-cloud abstraction libraries and dedicated cloud interfaces we described above also adopted the REST architecture, and therefore we also made our management interface comply with the RESTful style and design.

The structure of the implemented RESTful API web methods is shown in **Fig. 3** with some custom syntactic expressions. A service request can include additional options as query parameters or payloads. One example is creating a new instance as follows:

- HTTP verb and URI

```
POST http://localhost:4567/instances
```

- Payloads (URL encoded)

```
image_id=img-53c8800b&size_id=size-417719c2&number=1
```

Rebooting an instance is another example:

- HTTP verb and URI

```
PUT http://localhost:4567/instances/inst-41f7caf9/reboot
```

Note that the POST verb is used for creation, and the PUT verb is used for modification of a resource state. The most frequently used HTTP verbs, GET, POST, PUT and DELETE, have corresponding create, read, update and delete (CRUD) operations, respectively, which enables *uniform interface*, one of the RESTful design principles. In our implementation, the GET verb is used for listing existing resources with details, the POST method is used for creating a new resource, the PUT method is used for performing an action with resources, and the DELETE verb is used for terminating or deleting a resource. The list of methods and resources supported by our implementation is shown in **Table 4**.

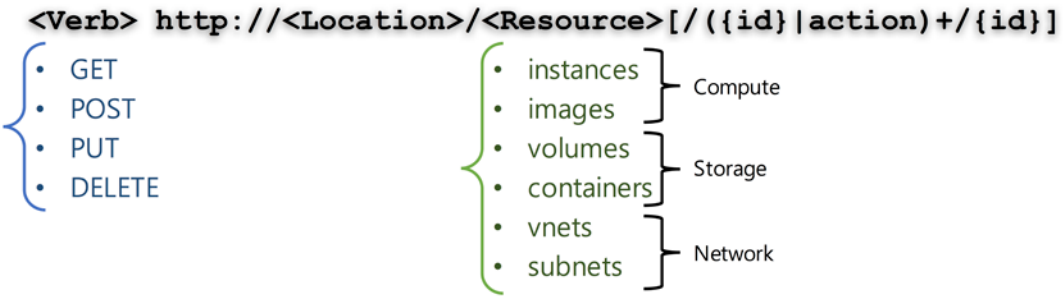


Fig. 3. Implemented RESTful API method structure

Table 4. RESTful API web methods

Resource type	***HTTP verb***	***Resource***	***Description***
Compute	GET	/sizes	Retrieve list and information of supported sizes
	GET	/images	Retrieve list and information of supported images
	GET	/instances	Retrieve list and information of running instances
	POST	/instances	Create a new instance with payloads
	PUT	/instances/{id}/reboot	Reboot an instance specified by {id}
	DELETE	/instances/{id}	Terminate an instance specified by {id}
Storage	GET	/volumes	Retrieve list and information of existing volumes
	POST	/volumes	Create a new volume with payloads

	PUT	/volumes/{id}/attach_to/{inst_id}	Attach a volume with {id} to an instance with {inst_id}
	PUT	/volumes/{id}/detach	Detach a volume with {id} from an instance
	DELETE	/volumes/{id}	Delete a volume specified by {id}
	GET	/containers	Retrieve list and information of existing containers
	POST	/containers	Create a new container with payloads
	DELETE	/containers/{id}	Delete a container specified by {id}
	GET	/objects	Retrieve list and information of uploaded objects
	POST	/objects/{filename}/upload_to/{cont_id}	Upload a file to a container with {cont_id}
	GET	/objects/{id}/download_from/{cont_id}	Download a file with {id} from a container with {cont_id}
	DELETE	/objects/{id}/from/{cont_id}	Delete a uploaded object in a container with {cont_id}
Network	GET	/vnets	Retrieve list and information of existing virtual networks
	POST	/vnets	Create a new virtual network with payloads
	DELETE	/vnets/{id}	Delete a virtual network specified by {id}
	GET	/subnets	Retrieve list and information of existing subnets
	POST	/subnets/in/{vnet_id}	Create a new subnet in a virtual network with {vnet_id} and payloads
	DELETE	/subnets/{id}	Delete a subnet specified by {id}

5.5 How It Works

A user or an application calls one of the REST API methods in a standard way with a resource name and options, and the web server gets the request and invokes an appropriate method that handles the request in the Integrated API. The method in turn validates and parses the input values, selects target cloud(s), and then calls a wrapper method in the target cloud(s) with the input values in order to perform the actual task. The wrapper method uses the ID conversion before calling a dedicated API method. The entry point information is used for the dedicated API method call, with which the method interacts with the API server of the target cloud.

The wrapper method gets the execution result from the dedicated API method and streamlines the data by extracting common and necessary values only. At this point, the local ID is converted to a global ID. A global ID is created only if it does not exist yet and returned from the ID map if it exists. The returned data are in turn converted to the XML format and delivered to the user or application.

6. Evaluation

In order to show the performance comparison between the individual cloud API calls and the integrated API calls, we measured total response time of each method on our testbed. The experiments are intended to reveal the overhead imposed by the integrated cloud API, not to evaluate the performance of each cloud API, and therefore the actual measured values are not a matter of interest, but the differences are of importance. API method calls and time measurement were performed using Ruby scripts, in which appropriate libraries (or Ruby Gems) were used such as REST Client for the RESTful API method call and built-in Benchmark module for the time measurement. Note that the response time does not necessarily mean the total elapsed time to complete a resource provisioning.

The experimental results are presented in **Table 5**. The blanks in the dedicated APIs column mean that the clouds in our testbed do not support those type of resources, for example, the CloudStack cloud does not support the object storage and virtual networking, and the OpenStack cloud partially supports the network resource type. The limited networking feature of our OpenStack cloud was supplemented by our implementation, but it was excluded for the measurement because it is not a part of the dedicated API. Similarly, the time for List Sizes operation for AWS could not be measured because the AWS API does not officially support a programmatic way to get the instance types it provides. We implemented the feature by ourselves. In the implemented API column, the time measurement is divided into two parts: time for operations that target each cloud individually and time for collective operations such as listing resources. Our API is cloud-agnostic as mentioned in the previous section, but we can indirectly designate a target cloud, for example by using a size or an image that is supported by the target cloud only in the case of Create an Instance. On the other hand, the listing operations retrieve list and information of all the resources of a certain type on the cloud testbed collectively and therefore can't be used separately. In the case of object storage and network resources, the target cloud can be chosen by a certain scheme, which is random in the current implementation, so it is not possible to choose the target cloud arbitrarily.

Fig. 4, 5, 6 and **7** show the results obtained for listing operations and operations on OpenStack, CloudStack, and AWS, respectively, which compare the response times measured from the dedicated APIs and the implemented API. As mentioned above, the listing operations get the information of each resource collectively from all clouds, the comparison was performed with the average values of the response times measured from the two sets of APIs. Instance- and volume-related operations were compared one on one and the results obtained for object storage resources were compared between method calls to AWS because only AWS provides the resource type in our testbed. The response times normally tend to increase in most cases due to the overhead imposed by the integrated API, but in some cases, the measured values decreased even though the integrated API plays a sort of wrapper role for the original APIs, so the results seem not to be correct. We suspect this is due to the network latency and the negligibly small overhead. Such cases mostly occurred in the operations on AWS that is located in the external network unlike other local private clouds, which supports our conjecture. Another possibility is that the overhead is so small that it may be within the margin of latency, and thus its impact is hidden from the final total response time. Indeed, the integrated API does not have so many functionalities as the dedicated APIs do, just collectively forwards and translates a request to access and manage a cloud resource into a suitable format for the target cloud and get the response message to show or transfer to other services. Therefore, the overhead imposed by the integrated API can be negligibly small.

Table 5. Experimental results (seconds)

	Dedicated APIs			*Implemented API*			
	OpenStack	CloudStack	AWS	OpenStack	CloudStack	AWS	(Collective)
List Sizes	0.017	0.013	-	-	-	-	0.045
List Images	0.042	0.020	0.065	-	-	-	0.144
List Instances	0.052	0.022	0.274	-	-	-	0.716
Create an Instance	0.493	0.454	0.853	0.559	0.448	0.890	-
Reboot an Instance	0.221	0.137	0.554	0.217	0.148	0.278	-
Terminate an Instance	0.286	0.134	0.495	0.295	0.145	0.464	-
List Volumes	0.060	0.016	0.445	-	-	-	0.485
Create a Volume	0.339	0.250	0.497	0.363	0.285	0.659	-
Attach a Volume	0.370	0.134	0.297	0.384	0.144	0.396	-
Detach a Volume	0.259	0.136	0.291	0.378	0.176	0.240	-
Delete a Volume	0.127	0.585	0.178	0.126	0.430	0.165	-
List Containers	-	-	0.703	-	-	-	0.520
Create a Container	-	-	1.429	-	-	-	1.146
Delete a Container	-	-	1.019	-	-	-	1.169
List Objects	-	-	0.701	-	-	-	1.283
Upload an Object	-	-	0.680	-	-	-	0.639
Download an Object	-	-	0.019	-	-	-	0.026
Delete an Object	-	-	0.600	-	-	-	0.666
List Virtual Networks	-	-	0.521	-	-	-	0.332
Create a Virtual Network	-	-	0.129	-	-	-	0.148
Delete a Virtual Network	-	-	0.469	-	-	-	0.304
List Subnets	0.013	-	0.495	-	-	-	0.604
Create a Subnet	0.366	-	0.150	-	-	-	0.141
Delete a Subnet	0.112	-	0.131	-	-	-	0.130

Fig. 4. Results of listing operations

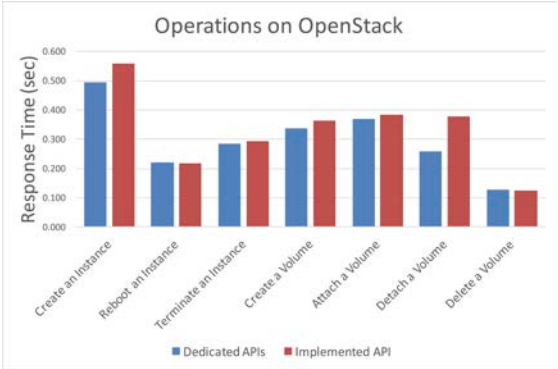


Fig. 5. Results of operations on OpenStack

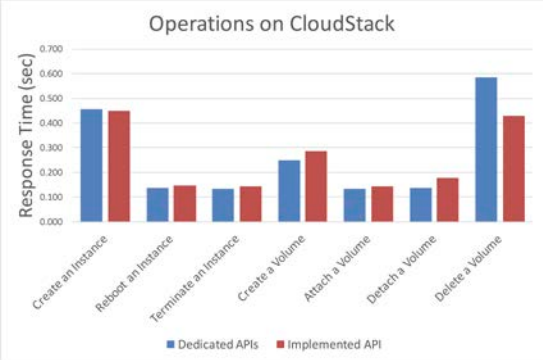


Fig. 6. Results of operations on CloudStack

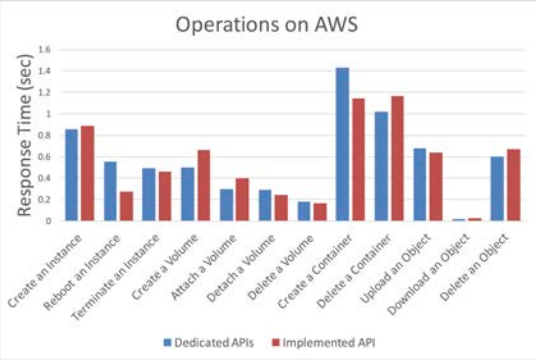


Fig. 7. Results of operations on AWS

7. Conclusions

7.1 Summary

The number of clouds per person is increasing, risks to solely rely on a single cloud service are getting higher, and users and applications are required to have the capability to interact with a multiple number of clouds more and more. To remove the heterogeneity and to facilitate the interoperable use of such multi-clouds, integrated and uniform accessibility to the multi-clouds is indispensable. In this paper, we surveyed existing interoperable solutions for multi-clouds including cloud interface standards, multi-cloud abstraction libraries and previous studies on integrated management of clouds.

Through the survey, we realized the issues and were inspired by the existing solutions, and finally we decided to develop our own integrated and uniform cloud management API for multi-clouds, due to the lack of some required features for our purpose: cloud-agnostic and simplicity for the future use of another research.

In order to develop our cloud abstraction API and evaluate it, we built a cloud testbed that consisted of two local private clouds, OpenStack and CloudStack, and the remote public cloud, AWS, provided by the cloud market leader, Amazon. We also identified the basic cloud resource types and defined common terms for the resources to reduce the confusion caused by the difference of terminologies. Our implementation is based on the existing dedicated cloud APIs such as OpenStack RESTful API, CloudStack Ruby client, and AWS Ruby SDK as well as the RESTful web service framework, Sinatra. We have successfully developed an integrated API for the management of basic cloud resource types, Compute, Storage and Network, and RESTful web methods to provide the integrated API to the users or applications in a uniform way.

Performance evaluation was also conducted and the results showed that there was no significant degradation generated by the implemented API, but rather some results showed the total response times were reduced, which was supposedly due to the network latency at the moment of the experiments implying the possibility that the overhead was too small to have impact on the overall response time. This is mainly due to the streamlined functionality of our implementation.

7.2 Future Work

As we mentioned in previous sections, we will combine the implemented API with the workflow-based cloud interface we proposed previously in place of the existing abstraction API that can only support EC2-compatible clouds to provide easy-to-use and integrated cloud interface for non-EC2-compatible multi-clouds. To prove its usefulness, we are planning to compose various patterns of cloud management workflows as many as possible.

Another work to do is an improvement of the integrated API. It lacks advanced features provided by the cloud providers as well as portability, which is one of the key enabling technologies for the use of multi-clouds. In the experiments, we could choose a desired cloud by using a resource restricted to the target cloud such as size and image. This is, however, not desirable feature. Instead, any resource should be able to work on any cloud no matter what resource a user or an application chooses to use, which is called cloud portability. We will study on the enabling technologies for the cloud portability such as Open Virtualization Format (OVF), and we will try to apply the technologies to our implementation and the testbed if needed.

References

[1] R. Buyya, C. S. Yeo, S. Venugopal, J. Broberg, and I. Brandic, "Cloud computing and emerging IT platforms: Vision, hype, and reality for delivering computing as the 5th utility," *Future Generation Computer Systems*, vol. 25, no. 6, pp. 599–616, Jun. 2009. Article (CrossRef Link)

[2] Yogesh Simmhan, Catharine van van Ingen, Girish Subramanian, and Jie Li, "Bridging the Gap between Desktop and the Cloud for eScience Applications," in *Proc. of 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pp. 474–481, 2010. Article (CrossRef Link)

[3] F. Hu *et al.*, "A Review on Cloud Computing: Design Challenges in Architecture and Security," *CIT. Journal of Computing and Information Technology*, vol. 19, no. 1, pp. 25–55, May 2011. Article (CrossRef Link)

[4] M. Keller, D. Meister, A. Brinkmann, C. Terboven, and C. Bischof, "eScience Cloud Infrastructure," in *Proc. of 2011 37th EUROMICRO Conference on Software Engineering and Advanced Applications (SEAA)*, pp. 188–195, 2011. Article (CrossRef Link)

[5] J. L. Hellerstein, K. J. Kohlhoff, and D. E. Konerding, "Science in the Cloud: Accelerating Discovery in the 21st Century," *IEEE Internet Computing*, vol. 16, no. 4, pp. 64–68, 2012. Article (CrossRef Link)

[6] "Amazon Web Services (AWS) - Cloud Computing Services," *Amazon Web Services, Inc.* [Online]. Available: https://aws.amazon.com/. [Accessed: 02-Dec-2016].

[7] "Google Compute Engine," *Google Cloud Platform*. [Online]. Available: https://cloud.google.com/compute/. [Accessed: 02-Dec-2016].

[8] "Apple iCloud," *Apple*. [Online]. Available: http://www.apple.com/icloud/. [Accessed: 02-Dec-2016].

[9] "Microsoft Azure: Cloud Computing Platform & Services." [Online]. Available: https://azure.microsoft.com/en-us/. [Accessed: 02-Dec-2016].

[10] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner, "A break in the clouds: towards a cloud definition," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 39, no. 1, pp. 50–55, Jan. 2009. Article (CrossRef Link)

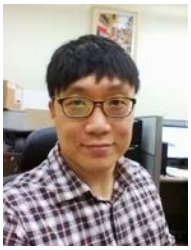
[11] Borja Sotomayor, Rubén S. Montero, Ignacio M. Llorente, and Ian Foster, "Virtual Infrastructure Management in Private and Hybrid Clouds," *IEEE Internet Computing*, vol. 13, no. 5, pp. 14–22, Oct. 2009. Article (CrossRef Link)

[12] P. M. Mell and T. Grance, "The NIST Definition of Cloud Computing," *National Institute of Standards & Technology*, Gaithersburg, MD, United States, SP 800-145, 2011. Article (CrossRef Link)
[13] Rafael Moreno-Vozmediano, Rubén S. Montero, and Ignacio Martín Llorente, "IaaS Cloud Architecture: From Virtualized Datacenters to Federated Cloud Infrastructures," *Computer*, vol. 45, no. 12, pp. 65–72, Dec. 2012. Article (CrossRef Link)
[14] R. Buyya, R. Ranjan, and R. N. Calheiros, "InterCloud: Utility-Oriented Federation of Cloud Computing Environments for Scaling of Application Services," in *Proc. of Algorithms and Architectures for Parallel Processing*, C.-H. Hsu, L. T. Yang, J. H. Park, and S.-S. Yeo, Eds. Springer Berlin Heidelberg, pp. 13–31, 2010. Article (CrossRef Link)
[15] Global Inter-Cloud Technology Forum, "Use Cases and Functional Requirements for Inter-Cloud Computing," Global Inter-Cloud Technology Forum, White Paper, Aug. 2010. Available: http://www.ttc.or.jp/files/8614/1214/5480/GICTF_Whitepaper_20100809.pdf. [Accessed: 02-Dec-2016].
[16] A. N. Toosi, R. N. Calheiros, and R. Buyya, "Interconnected Cloud Computing Environments: Challenges, Taxonomy, and Survey," *ACM Computing Surveys*, vol. 47, no. 1, pp. 1–47, May 2014. Article (CrossRef Link)
[17] N. Grozev and R. Buyya, "Inter-Cloud architectures and application brokering: taxonomy and survey," *Softw. Pract. Exper.*, vol. 44, no. 3, pp. 369–390, Mar. 2014. Article (CrossRef Link)
[18] T. Metsch and A. Edmonds, "Open Cloud Computing Interface - RESTful HTTP Rendering." Open Grid Forum, 21-Jun-2011. Available: https://www.ogf.org/documents/GFD.185.pdf. [Accessed: 02-Dec-2016].
[19] Jacques Durand, Marios Andreou, Doug Davis, and Gilbert Pilz, Eds., "Cloud Infrastructure Management Interface (CIMI) Model and RESTful HTTP-based Protocol." Distributed Management Task Force, 20-Mar-2015. Available: http://www.dmtf.org/sites/default/files/standards/documents/DSP0263_2.0.0c.pdf. [Accessed: 02-Dec-2016].
[20] "OASIS Topology and Orchestration Specification for Cloud Applications (TOSCA)." [Online]. Available: https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=tosca. [Accessed: 02-Dec-2016].
[21] "Deltacloud API." [Online]. Available: https://deltacloud.apache.org/. [Accessed: 02-Dec-2016].
[22] "Apache jclouds." [Online]. Available: https://jclouds.apache.org/. [Accessed: 02-Dec-2016].
[23] "Apache Libcloud," *Apache Libcloud*. [Online]. Available: https://libcloud.apache.org/. [Accessed: 02-Dec-2016].
[24] F. Meireles and B. Malheiro, "Integrated Management of IaaS Resources," in *Proc. of Euro-Par 2014: Parallel Processing Workshops*, L. Lopes, J. Žilinskas, A. Costan, R. G. Cascella, G. Kecskemeti, E. Jeannot, M. Cannataro, L. Ricci, S. Benkner, S. Petit, V. Scarano, J. Gracia, S. Hunold, S. L. Scott, S. Lankes, C. Lengauer, J. Carretero, J. Breitbart, and M. Alexander, Eds. Springer International Publishing, pp. 73–84, 2014. Article (CrossRef Link)
[25] T. Harmer, P. Wright, C. Cunningham, and R. Perrott, "Provider-Independent Use of the Cloud," in *Proc. of Euro-Par 2009 Parallel Processing*, H. Sips, D. Epema, and H.-X. Lin, Eds. Springer Berlin Heidelberg, pp. 454–465, 2009. Article (CrossRef Link)
[26] Thijs Metsch, Andy Edmonds, and Victor Bayon, "Using Cloud Standards for Interoperability of Cloud Frameworks," SLA@SOI, Technical Report. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.462.5624. [Accessed: 02-Dec-2016].
[27] N. Loutas, V. Peristeras, T. Bouras, E. Kamateri, D. Zeginis, and K. Tarabanis, "Towards a Reference Architecture for Semantically Interoperable Clouds," in *Proc. of 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom)*, pp. 143–150, 2010. Article (CrossRef Link)
[28] A. Celesti, F. Tusa, M. Villari, and A. Puliafito, "How to Enhance Cloud Architectures to Enable Cross-Federation," in *Proc. of 2010 IEEE 3rd International Conference on Cloud Computing (CLOUD)*, pp. 337–345, 2010. Article (CrossRef Link)

[29] N. M. Calcavecchia, A. Celesti, and E. Di Nitto, "Understanding Decentralized and Dynamic Brokerage in Federated Cloud Environments," in *Proc. of Achieving Federated and Self-Manageable Cloud Infrastructures: Theory and Practice*, IGI Global, 2012. Article (CrossRef Link)
[30] G. Arunkumar and N. Venkataraman., "A Novel Approach to Address Interoperability Concern in Cloud Computing," *Procedia Computer Science*, vol. 50, pp. 554–559, 2015. Article (CrossRef Link)
[31] Á. López García, E. Fernández del Castillo, and P. Orviz Fernández, "Standards for enabling heterogeneous IaaS cloud federations," *Computer Standards & Interfaces*, vol. 47, pp. 19–23, Aug. 2016. Article (CrossRef Link)
[32] L. Heilig, E. Lalla-Ruiz, and S. Voß, "A cloud brokerage approach for solving the resource management problem in multi-cloud environments," *Computers & Industrial Engineering*, vol. 95, pp. 16–26, May 2016. Article (CrossRef Link)
[33] "rOCCI - A Ruby OCCI Framework," *GitHub*. [Online]. Available: https://github.com/gwdg/rOCCI. [Accessed: 02-Dec-2016].
[34] "Service Sharing Facility." [Online]. Available: http://pyssf.sourceforge.net/. [Accessed: 02-Dec-2016].
[35] "erocci." [Online]. Available: http://erocci.ow2.org/ - !/main. [Accessed: 02-Dec-2016].
[36] "OpenStack Open Source Cloud Computing Software." [Online]. Available: https://www.openstack.org/. [Accessed: 02-Dec-2016].
[37] "Apache Cloudstack," *Apache Cloudstack*. [Online]. Available: https://cloudstack.apache.org/. [Accessed: 02-Dec-2016].
[38] B. Rochwerger *et al.*, "The RESERVOIR model and architecture for open federated cloud computing," *IBM Journal of Research and Development*, vol. 53, no. 4, p. 4:1-4:11, Jul. 2009. Article (CrossRef Link)
[39] "The SLA at SOI project." [Online]. Available: http://sla-at-soi.eu/. [Accessed: 02-Dec-2016].
[40] Kevin Kelly, "The Technium: A Cloudbook for the Cloud." [Online]. Available: http://kk.org/thetechnium/a-cloudbook-for/. [Accessed: 02-Dec-2016].
[41] H. Kim, K. Chun, H. Kim, and Y. Chung, "Utilization of workflow management system for virtual machine instance management on cloud," *Concurrency Computat.: Pract. Exper.*, vol. 27, no. 17, pp. 5350–5373, Dec. 2015. Article (CrossRef Link)
[42] "Thin - yet another web server." [Online]. Available: http://code.macournoyer.com/thin/. [Accessed: 02-Dec-2016].
[43] "Sinatra." [Online]. Available: http://www.sinatrarb.com/. [Accessed: 02-Dec-2016].
[44] "OpenStack API Documentation." [Online]. Available: http://developer.openstack.org/api-guide/quick-start/index.html. [Accessed: 02-Dec-2016].
[45] "Cloudstack ruby client." [Online]. Available: https://chipchilders.github.io/cloudstack_ruby_client/. [Accessed: 02-Dec-2016].
[46] "AWS SDK for Ruby." [Online]. Available: https://aws.amazon.com/sdk-for-ruby/. [Accessed: 02-Dec-2016].
[47] R. T. Fielding, "Architectural Styles and the Design of Network-based Software Architectures," University of California, Irvine, 2000. Available: https://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm. [Accessed: 02-Dec-2016].

Huioon Kim received the B.S. degree in Computer Engineering from Dong-a University, Korea, in 2005 and M.S. degree in Information and Communications from Gwangju Institute of Science and Technology, Korea, in 2008. He is currently a Ph.D. candidate in the School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology, Korea. His research interests include cloud computing, eScience research environment and high-performance computing.

Hyounggyu Kim received the B.S. and M.S. degrees in Physics from Sejong University, Korea, in 2006 and 2008, respectively. He is currently a Ph.D. candidate in the School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology, Korea. His research interests include particle simulations using cloud computing, high-performance computing based on GPGPU and system integration.



Kyungwon Chun received the B.S. degree in Physics from Chung-Ang University, Korea, in 2001 and the M.S. and Ph.D. degrees in Information and Communications from Gwangju Institute of Science and Technology, Korea, in 2004 and 2013, respectively. His research interests include numerical simulations, high-performance computing and programming algorithms.

Youngjoo Chung received the B.S. degree in Physics from Seoul National University, Korea, in 1982 and the M.S. and Ph.D. degrees in Plasma Physics from Princeton University, USA, in 1985 and 1989, respectively. He is currently a full professor of the School of Electrical Engineering and Computer Science, Gwangju Institute of Science and Technology, Korea. His research interests include Computer-aided mathematics, mathematical analysis based on high-performance computing and E-education for mathematics.