Cluster Comput (2016) 19:1585–1597
DOI 10.1007/s10586-016-0599-0

An efficient multi-task PaaS cloud infrastructure based on docker and AWS ECS for application deployment

Gemoh Maliva Tihfon[1] · **Sanghyun Park**[1] · **Jinsul Kim**[1] · **Yong-Min Kim**[2]

Received: 14 June 2016 / Accepted: 4 July 2016 / Published online: 23 July 2016

Abstract The setup environment and deployment of distributed applications is a human intensive and highly complex process that poses significant challenges. Nowadays many applications are developed in the cloud and existing applications are migrated to the cloud because of the promising advantages of cloud computing. Presenting two common serious challenging scenarios in the application development environment, we propose a multi-task PaaS cloud infrastructure using Docker and AWS services for application isolation, optimization and rapid deployment of distributed applications. We fully utilized Docker, a lightweight containerization technology that uses a host of the Linux kernel's features such as namespaces and cgroup's to sandbox processes into configurable virtual environments. The Amazon EC2 container service helps our container management framework. The cluster management framework uses optimistic, shared state scheduling to execute processes on EC2 instances using Docker containers. Several experimentations were carried out, one of the experimentation focused on a simulation of application deployment scheduling that shows our propose infrastructure is flexible, efficient and well optimized.

Keywords App deployment · Cloud computing · Virtualization · Hypervisors · Virtual machine (VM) · Container · Clustering · Docker

1 Introduction

Software deployment is complex and the diverse computing requirements for applications require complex hardware infrastructure setups and possibly incompatible specific software requirements. Therefore, a platform to automate the deployment and setup of virtual computing is essential. Moreover, it is important to properly and efficiently manage the computing resources so as to reduce additional investment in hardware. All these lead towards the concept of cloud computing. Cloud computing is a paradigm to rapidly provision computing resources such as storage, networks, servers, services, etc, that can be customized and configured to suit a particular user or application demands [1,2]. Cloud computing paradigm is promising because is changing the way enterprises do their businesses in that dynamically scalable and virtualized resources are provided as a service over the internet. The cloud is enabled by virtualization, automation, standardization. The very core of cloud computing is virtualization, which is used to separate a single physical machine into multiple virtual machines in a cost-effective way. By using virtualization, we're basically getting a lot of the work done for free. With virtualization, a number of virtual machines can run on the same physical computer, which makes it cost-effective, since part of the physical server resources can also be leased to other tenants [3,4]. Such vir-

✉ Jinsul Kim
jsworld@chonnam.ac.kr

✉ Yong-Min Kim
ymkim@chonnam.ac.kr

Gemoh Maliva Tihfon
gemohmal@gmail.com

Sanghyun Park
sanghyun079@gmail.com

[1] School of Electronics and Computer Engineering, Chonnam National University, Gwangju, Korea

[2] Division of Culture Contents, Chonnam National University, Yeosu, Korea

tual machines are also highly portable, since they can be moved from one physical server to the other in a manner of seconds and without downtime; new virtual machines can also be easily created. Another benefit of using virtualization is the location of virtual machines in a data center. It does not matter where the data center is located and the virtual machine can also be copied between the data centers with ease [5]. VM's in the cloud offer rapid elasticity and it is pay as you go model. One thing to keep in mind before pushing forward is that, it's all about the applications and not the operating system. We need operation system to facilitate the applications. Virtual machines (VM) reduce capital expenditure and operational expenditure but also have some limitations that can be address. VM still requires a CPU, storage allocation, RAM, and an entire guest Operating system (OS). OS consume a lot CPU, RAM, Disk storage, and increase overhead. The more VM's you run, the more resources you need. Also some operating systems might need individual licensing. Moreover application portability is not guaranteed. The VM module does nothing to help but with Docker, we don't have to worry about all the issues mentioned above. IaaS cloud computing is hugely influence by hypervisor virtualization [6]. Lightweight virtualization technologies such as Docker, LXC, and Open VZ etc, seems to be a good fit for the cloud although lightweight virtualization is limited but they provide a better hosting density. In general it is possible to host more lightweight virtualizations on a physical host than with hypervisor based virtualizations [7,8]. Docker can be deployed in any environment or device being public or private cloud because it is super lightweight. A Docker container does not include the full OS as mention earlier, but shares the OS of its host. As a result, Docker containers can be faster and less resource-draining than virtual machines. Isolation of resources is a good fit for virtual machine but to run hundreds of isolated processes on an average host, Docker is the better fit. Docker is a good tool for development, QA, system admins, performance environments on old hardware. A full virtual machine can take a couple of minutes to launch because of boot time and other stuff, however a container can be initiated in a blink of an eye (seconds). Containers also offer superior performance for the application they contain, compared to running the application within a virtual machine. In this paper, we propose an efficient environment for application deployment that combines Docker and AWS ECS to produce a simple and yet optimized cloud infrastructure. Our paper is made possible by Docker that has gained widespread popularity in recent years. The rest of our paper is organized in the following order. Section 2 will state the problems and challenges of application development and deployment. Also in this section is details understanding of Docker containerization. Section 3 presents the related works. In Sect. 4, we present our propose cloud platform implementation and discuss the working flow and also the major components of it. Following up is Sect. 5 with our experimentations and results. And finally in Sect. 6, we conclude the paper and introduce further future research work.

2 Problem statement and background

Scenario 1 Microservice architecture is an approach that makes web based development more agile and code bases easier to maintain. This architecture enables developers to be highly productive and to quickly iterate and evolve a code base. For fast moving startup companies, the microservices architecture can really help dev teams be quick and agile in their development efforts. The disadvantage of microservices is that, because services are spread across multiple hosts, it is difficult to keep track of which hosts are running certain services. Docker containers can help mitigate many of these challenges with the microservices architecture. Docker containers make use of kernel interfaces such as cgroups, namespaces, and union files, which allow multiple containers to share the same kernel while running in complete isolation from one another. The Docker execution environment uses a module called Libcontainer, which standardizes these interfaces. It is this isolation between containers running on the same host that makes deploying microservices code developed using different languages and frameworks very easy. Using Docker, we could create a DockerFile describing all the languages, framework, and library dependencies for that service. The container execution environment isolates each container running on the host from one another, so there is no risk that the language, framework, library dependencies used by one certain container will collide with that of another.

Scenario 2 You have written a code for some website or developed a mobile app for a game using development environment on your laptop. After thorough testing and realize that your code is ready to be deploy in the working environment or in your working organization. The system admin dutifully deploys the most recent build to the test environment and in no time notice that your recently developed REST endpoint is broken. After uncountable hours of troubleshooting with the system admin, you discover that the test environment is using an outdated version of third-party library, and this was causing the REST endpoints to break. Differences between developments, test, stage, and production environments is a familiar problem in today's rapid build and deploy cycles. The solution is to find a way to transfer from one environment to another seamlessly and eliminating error prone resource provisioning and configuration. Services like Amazon EC2, AWS CloudFormation, and Docker provide reliable and efficient way to automate the creation of an environment.

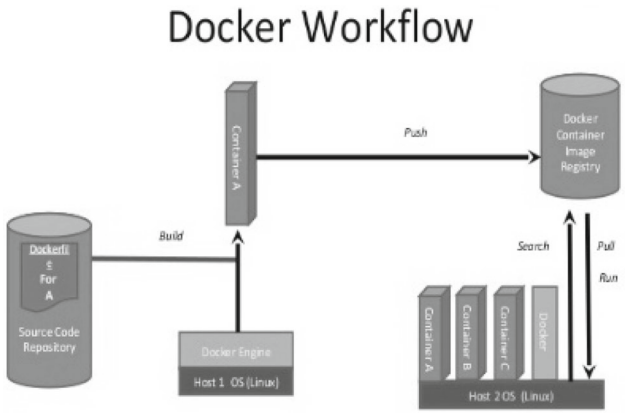

Fig. 1 Docker workflow diagram

Amazon EC2 makes web-scale cloud computing easier for developers. AWS CloudFromation gives developers and system admins an easy way to create and manage a collection of related AWS resources, provisioning and updating them in an orderly and predictable manner [9–11]. You simply create or use prebuilt template which is a JSON file that serves as a blueprint to define the configuration of all the AWS resources that make up your structure and application stack. On a plus, CloudFormation is free of charge and you pay only for the AWS resources needed to run your application. Docker takes the concept of declarative provisioning a step further. Docker provides a declarative syntax for creating containers. However, Docker containers don't depend on any specific virtualization platform; neither do they need a separate operating system to run. A container simply requires a Linux kernel in order to run [12,13]. This means dockerized apps can run anywhere on anything being desktop, laptops, VMs, datacenter or instances in the cloud. Docker containers use an execution environment called Libcontainer, which is an interface to various Linux kernel isolation features, like cgroups, namespace, and union files. This architecture allows for multiple containers to be run in complete isolation from one another while sharing the same Linux kernel. Because a Docker container instance doesn't require a dedicated OS, it is much more portable and lightweight than a virtual machine. The core components of Docker are the Docker daemon and Docker client. Docker daemon is the engine that runs on the host machine and it is a server process that manages all the containers. Docker client is a CLI used to interact with the daemon. The key concepts to understand the workflow of Docker as shown in Fig. 1 are its workflow components. Docker images, registry, containers, and Dockerfile.

- Docker image holds the build component of a container. It is a read-only template from which container instances can be launched. Think of it as Amazon AMI.
- Docker registry or DockerHub is a public and private repositories used to store images. It is use to distribute images efficiently and securely.
- Docker container is a running instance of an image or created from images. Docker uses containers to execute and run (start, stop, move, delete) the software contained in the image. We can create Docker images from a running container, similar to the way we create an AMI from an EC2 instance. For example, you could launch a container, install a bunch of software packages using a package manager like apt-get or yum, and then commit (save) those changes to a new Docker image.
- DockerFile is a more efficient and flexible way to create an image. It automates image construction.

Docker containers are becoming the go ahead for all distributed systems because they are scalable in the sense that these containers are extremely lightweight which make scaling up and scaling down very fast and easy. Dockerized applications are extremely portable; we can move them very easy. With the isolated containers, we can put more than one into a machine thereby making more efficient use of our resources. Another huge plus point of Docker is the Docker community. This community is one of the fastest growing open source communities out there. Chef, Puppet, Cloud providers such as AWS, OpenStack Azure, and Rackspace are just a few of the recognized members. There are many more benefits, but what all this mean is that it dramatically reduces the entire development life cycle from development, to testing, and then deployment.

2.1 Background of containers and docker

Operating system-level virtualization or nowadays call Containers is a server virtualization method where the kernel of an operating system allows for multiple isolated user space instances, instead of just one. Such instances (often called containers, virtualization engines (VE), virtual private servers (VPS), may look and feel like a real server from the point of view of its owners and users. In addition to isolation mechanisms, the kernel often provides resource management features to limit the impact of one container's activities on the other containers. In contrast to containers is a hypervisor virtualization (VMware, Hyper-V etc.). Each container has its own root file system, processes, memory, devices, and network ports or stacks. Docker is technically a Linux container because it uses almost all of the Linux kernel's features such as namespaces, cgroups, UnionFS, AppArmor profiles [12]. Namespace provides a layer of isolation. Each aspect of a container runs in its own namespace and does not have access outside it. Control groups (cgroups) provide resource

management. In addition to isolation mechanisms, docker uses cgroups to provide resource management to limit the impact of one container's activities on the other containers. Union File Systems (UnionFS) are file systems that operate by creating layers, thus making them very lightweight and fast. Docker also uses UnionFS to provide a building block for containers. Later we will see hoe with tag image support on docker, we update our applications just by downloading a layer instead of the whole application. Docker uses and controls its own execution driver called libcontainer (default container format) [14]. The libcontainer is use to group all the Linux kernel features together. They don't rely on traditional LXC and this means that they can go cross platform.

3 Related works

There are many works in software management and tools that address the deployment of virtual infrastructures and applications. Numerous cloud providers, such as AWS provide tools to deploy virtual infrastructures, applications and websites. In particular, CloudFormation and OpsWorks provides developers and systems administrators with an easy way to create and manage a collection of related AWS resources, provisioning and updating them in an orderly and predictable fashion [15]. The Nimbus project team group has developed a set of tools to deploy virtual infrastructures: the Context Broker [16,17] and cloudinit.d [18]. In particular, the last tool submits, controls, and monitors Cloud applications. It automates the virtual machine (VM) creation process, the contextualization, and the coordination of service deployment [19,20]. It supports multiple clouds and the synchronization of different 'runlevels' to launch services in a defined order. Furthermore, it provides a system to monitor the services that uses user-created scripts to ensure that they are running. This system checks for service errors, re-launching failed services or launching new VMs. However, it enables the contextualization of VMs using simple scripts, which are insufficient in complex scenarios with multiple VMs and different Operating Systems. One common limitation of all the above systems is the usage of manually selected base images to launch the VMs. This is an important limitation because it implies that users must create their own images or they must previously know the details about software and configuration of the image selected. This limitation affects the reutilization of the previously created VMIs, forcing the user to waste time testing the existing images or creating new ones. Another issue is that most of them need to use a VMI specifically configured to support their environment, requiring specific software installed or a set of scripts prepared. The next section is our proposed cloud platform tries to address and improve most of these related works. Also in the next section is a scheduling algorithm we created to effectively and fully utilize the available resources in any data center or private organization setup environment. Following up is the experiment and evaluation of our work and then conclusion of this paper.

4 Proposed cloud infrastructure

Based on the numerous advantages of Docker containers, ease of deployment in the development, test, stage, and production environment and how Docker containers fit well in the distributed systems architecture (microservices). We propose a multi-task PaaS cloud system which is shown in Fig. 2. This PaaS cloud system using Docker is for infrastructure virtualization and application isolation/deployment. Applications are develop using Docker technology and distributed to the end users efficiently with AWS EC2 services. The propose platform allow organizations or developers to focus on building products rather than building infrastructure. Developers can design any web site or mobile application effortlessness using their language of preference and on any device based on this infrastructure. However, in a multi-task environment, the number of containers will be increasing, and this becomes increasingly difficult to manage manually. This is where the services of Amazon EC2 Container Service (ECS) steps in to help our container management framework (cluster computing). With ECS, we effectively abstract the low-level resources such as CPU, memory, and storage, allowing for highly efficient usage of the nodes in a compute cluster. Initially the idea we had was to use the Docker swarm which is a native clustering solution provided by Docker. It takes the Docker Engine and extends it to enable you to work on a cluster of containers. Using Swarm we can manage a resource pool of Docker hosts and schedule containers to run transparently on top, automatically managing workload and providing failover services. Swarm uses an algorithm called Bin Packing Scheduling algorithm (they also support Random and Spread algorithms) and some scheduler filters (Constrain, Affinity, Port, Dependency, and Health filters) to effectively manage the containers on a subset of nodes. Swarm is the future native clustering for Docker. Currently swarm has many limitations such as it doesn't support image management yet, it is still beta and not recommended for production. So using Amazon EC2 Container service is the right choice for scalability and management of Docker containers. EC2 Container Service is a cluster management framework that uses optimistic, shared state scheduling to execute processes on EC2 instances using Docker containers. Amazon ECS makes it easy to launch containers across multiple hosts, isolate applications and users, and scale rapidly to meet changing demands of your applications and users.

Using ECS incurs no extra charges, apart from the cost of spinning up EC2 instance. The ECS takes care of many of the challenges in running a distributed system. Customers need not about monitoring the health and availability of nodes that provide the scheduling and resource management capabilities. ECS also provides a robust solution to the very challenging problem of storing state information in a distributed system. Lastly, ECS is designed to scale horizontally and for high availability. Container instances, clusters, tasks, and task definition are the key components of ECS.

Our propose platform allow organizations/sysadmins/developers to focus on building products rather than building infrastructure. As mention earlier, we can build, test, and debug our code on any machine capable of running Docker. When the code is ready, we package it up into the Docker image by building the image from a Dockerfile and storing it in Docker Hub (repository). Next, we provide the compute resources required to run containers using Amazon ECS or the schedule algorithm used for a more private and secure platform environment. In ECS, this is called a cluster, and it consists of EC2 instances called "container instances" that are running the ECS agent. To create an ECS cluster of container instances, we simply launch one or more EC2 instances using the Amazon ECS-Optimized Amazon Linux AMI. The instance will need to be associated with an IAM role that allows the agent running on the instance to make the necessary API calls to ECS. The next step is to tell ECS how to run the containers. We use an entity called a "task definition." ECS task definition can be thought of a prototype for running an actual task. For any given task definition, there can be zero or more task instances running in the cluster. The task definition allows for one or more containers to be specified. ECS has another entity called a "service," which is useful for long running tasks, like web applications. The service allows multiple instances of a task definition to be run simultaneously. It also provides integration with Elastic Load Balancing (ELB) service. The ELB is used to distribute tasks and services to different containers efficiently.

To run a job, a developer simply needs to express the job, often through a config file or shell script as a collection of tasks and then submit the job to the scheduler for execution. The cluster management takes care of everything including check-pointing and re-queuing of a failed tasks. The cluster management framework can efficiently allocate resources and schedule tasks. The schedule algorithm below aims to schedule applications on VMs based on the user deployment request and deploys VMs on physical resources based on resource availability. This strategy optimizes the application performance. Additionally, the load-balancer ensures high and efficient resource utilization in the cloud environment.

Schedule Algorithm for VMs

```
1: Input: UserDeploymentRequest
2: get Resources&AvailableVMList
3: // find applicableVMList
4: if AVM(UDR, ARS) != 0 then
5:     //call the load balancer
6:     deployableVM = load-balance(AVM(UDR, ARS))
7:     deploy UserRequest on deployableVM;
8:     deployed = true;
9: else
10:    if ResourceForExtraVM   then
11:            start newVMInstance;
12:            add VMToApplicableVMList;
13:            deploy UserRequest on newVM;
14:            deployed = true;
15:    else
16:            queue UserRequest until
17:            queueTime > waitingTime
18:            deployed = false;
19:    end if
20: end if
21: if deployed   then
22:    return success;
23:    terminate;
24: else
25:    return fail;
26:    terminate;
27: end if
```

As shown in the Schedule Algorithm, the scheduler receives as input the Users' Deployment Requests (UDR) and the application data to be provisioned (line 1 in the schedule algorithm). The output of the scheduler is the confirmation of successful or failure deployment. In the first step, the user request is extracted, which then forms the basis for finding the VM with appropriate resources for deploying the application. Next, it collects information about the Available Resource (ARS) and the number of running VMs in the data center (line 2). The UDRs are used to find a list of Applicable/Apposite VMs (AVM) capable of provisioning the requested user request (lines 3–4). When the list of VMs is found, the load-balancer decides on which particular VM to deploy the application in order to balance the load in the data center; in our case the ELB (line 5–8). In case there is no VM with the appropriate resources running in the data center, the scheduler checks if there is resources consisting of physical resources can host new VMs (lines 9–10). If that is the case, it automatically starts new VMs with predefined resource capacities to provision the user requests (lines 11–14). When the resources cannot host extra VMs, the scheduler queues the provisioning of service requests until a VM with appropriate resources is available (lines 15–16). If after a certain period of time, the user requests cannot be be scheduled and deployed, the scheduler returns a scheduling failure to the

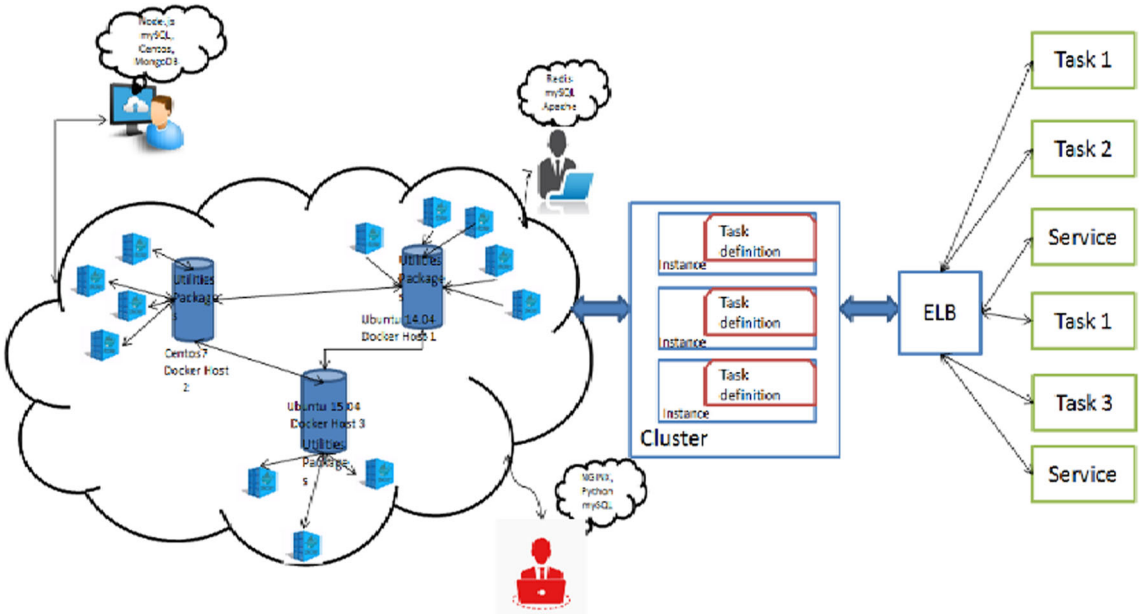

Fig. 2 Propose multi-task platform

cloud admin, otherwise it returns success (lines 17–27). The scheduler is also responsible for scheduling requests that are in the queue and allows for the use of cluster's idle resources to satisfy a user-request requirement. User-request priority in the queue is calculated based on the request attributes and the resources are analyzed before the request is being placed in the scheduler queue. The below equation shows requests with less priority that are preempted first.

$$P_r = W_1.(UXU - UAU) + W_2.Q_t + W_3.(ARS + AVM) + W_4.N_t + W_5.N_T + W_6.(TRS - (AVM - ARS)) + W_7.K \quad (1)$$

From Eq. 1 above, P_r is priority request, UXC is the user's expected usage, UAU is a user actual usage, Q_t is the time a user-request has spent in the queue, AVM is the available VMlist, ARS is the available resource, N_t is the number of time a request has been preempted, NT is the total number of times a request has been preempted, TRS is the total available resource, the request is active if K is 1 or 0 if inactive, $W_1, \ldots\ldots\ldots\ldots, W_7$ are weighting factors that are empirically determined and is the weighting factor to elevate active requests.

5 Experimentation and evaluation

Our experimentation and evaluations are divided into three parts. The first part of the experiment is using Docker and source code of a web application to create an image that can be deploy and run in any environment. The next part of our experimentation is a Docker container evaluation. The final part of our experimentation is focused on a simulation of application deployment scheduling.

5.1 Docker image experiment

Based on the figure below Fig. 3, the setup environment for testing is pretty simple at this stage of our work. Using Oracle VM VirtualBox manager we setup a 64-bit Ubuntu 14.04 system and Centos7 system with the following features each: 512MB of memory, 2 processor, 12MB of video memory, 2 network adapters, and 16GB of hard drive space.

First, we define a Docker image for launching a container for running the REST endpoint. We will then use this Docker image to test the code on the Centos7 (acting like a laptop in this test environment). Later this created image can be used to test the code in Amazon EC2. The REST endpoints are going to be developed using Ruby and the Sinatra framework, so these is needed to be installed in the image. Sinatra is open source software to write web application written in Ruby. We chose Sinatra framework in the test environment because it is an elegant web framework and really tiny (about 1500 line code). Sinatra is good fit for small scale projects and it does all what other heavy frameworks of the Ruby family such as Rail. The backend will use Amazon DynamoDB to ensure that the application can be run from both inside and outside AWS web services, the Docker image will include the DynamoDB local database. The Docker image is created using the DockerFile that contains all the instructions require to build an image. DockerFile is similar to the way we create AMI from an EC2 instance. From the file, we will launch

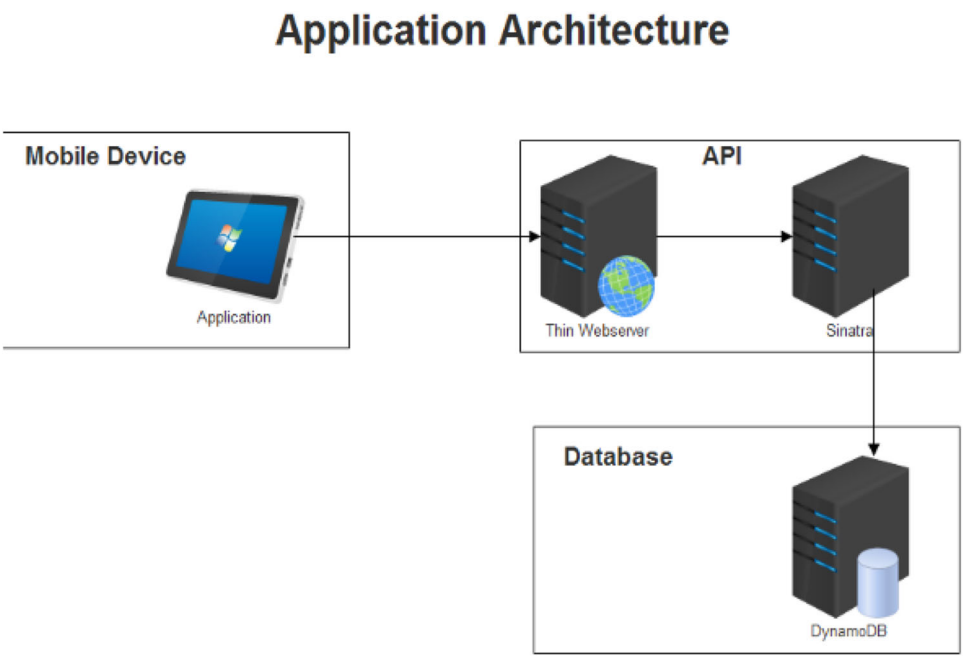

Fig. 3 A Web application architecture

containers, install a bunch of software packages using the APT package manager, and then commit those changes to a new Docker image. DockerFile is a more powerful, fast and flexible way of creating Docker images. Here's the DockerFile we created for the web app looks like:

To build the image from the above DockerFile, we used this command

$ docker build - -tag="aws_activate/sinatra:v1"

The tag option sets an identifier on the images and is usually setup as owner/repository:version. This makes it easy to identify what an image contains and who owns it when viewing the images in a registry. Next we launched a container from this newly created image:

$ docker run -it aws_activate/sinatra:v1 /bin/bash

This command launches the container and goes into a bash shell. We can interact with the container inside just like we would on a Linux server. Because we are developing a web application, we cloned the image and commit the changes in the running container to a new image using this commit command.

$ docker commit -m "ready for testing" b9d03d60ba89 aws_activate/sinatra:v1.1

Version 1.1 of the container includes the Sinatra application that will serve up the REST endpoint. The web application can be run using this command:

$ docker run -d -w /home/sinatra –p 10001:4567aws_ activate/sinatra:v1.1 ./run_app.sh

The shell script starts up the local DYnamoDb database in the container and launches the Sinatra application using the thin webserver on port 4567. The web application can be view from the browser using http://localhost:10001/activity/1 and see the following:

{"activity_id":"1", "user_id":" db430d35-92a0-49d6-ba79-0f37ea1b35f7", "type":"meal", "calories":100, "date": "2015-10-29 15:47:23 +0000"}

The endpoint seems to be working properly. The activity record was pulled from the local DynamoDB database and returned as JSON from the Sinatra application code.

5.2 Docker containers evaluation

To perform how efficient and less overhead the docker containers are, we created five different containers in detached mode (running in the background) and then attach back to some of the containers to install, run and updated some packages. From the pulled centos image, we installed traceroute, vim and created some files in the container. We also updated the image with a "yum update". In the pulled Ubuntu 14.04.1 image pulled from the docker hub, we installed these packages to run a simple apache web server; goland, nginx, apache2, apache2-utils, iputils-ping, and traceroute. We also ran an update. The created containers names were rename to easily check the different performance of the containers in the network.

Using the "docker stats" we were able to view the CPU usage, memory usage, memory limit, and network IO metrics of the different live stream containers at runtime as shown in Table 1. The results were outstanding when compared with VMs. This shows the operational benefits of docker containers and the density potential gained by using the docker container technology versus traditional VMs.

Table 1 Docker containers runtime metric

CONTAINER	CPU (%)	MEM USAGE/LIMIT	MEM (%)	NET I/O
Centos7	0.00	1.27 MiB/489 MiB	0.26	37.27MiB/580.5 KiB
mongodb	1.88	22.18MiB/489 MiB	4.54	1.307 KiB/648 B
nginx	0.00	1.23 MiB/489 MiB	0.25	1.939 KiB/648 B
redis	0.41	1004 KiB/489 MiB	0.22	2.572 KiB/648 B
Ubuntu14	0.00	1.785MiB/489MiB	0.37	28.9MiB/400.1KiB

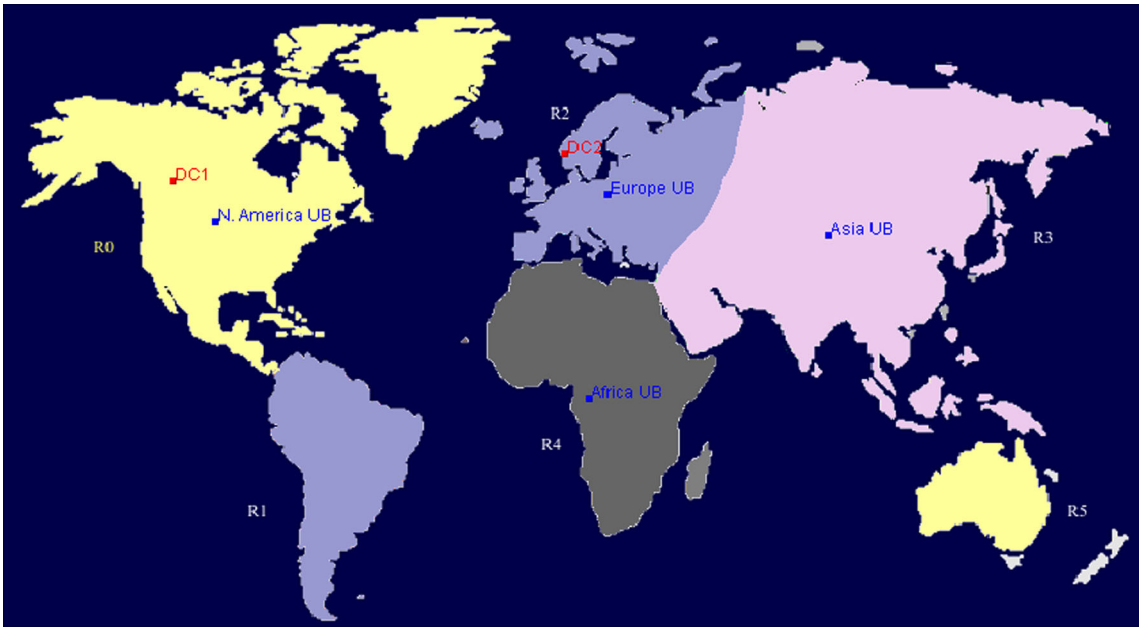

Fig. 4 Region and user base setup in our evaluation with web application

5.3 App deployment and scheduling simulation

Cloud aim to power the next-generation data centers as the enabling platform for dynamic and flexible application provisioning [21,22]. Scheduling is one of the most important task in cloud computing.

Using cloud as the application hosting platform, IT companies are freed from the trivial task of setting up basic hardware and software infrastructures. The use of real infrastructures (such as Amazon EC2, Google Cloud, Microsoft Azure) for benchmarking the application performance under different conditions is usually constrain. Simulated-based approaches offer significant benefits to IT companies by allowing them to test, tune and experiment with different workloads and resource performance scenarios on simulated infrastructures for developing and testing adaptive application provisioning techniques. CloudSim is a new generalized and extensible simulation framework that allows seamless modeling, simulation, and experimentation of emerging cloud computing infrastructures and application cervices [23]. Although several similar experimentation works has been carried out with CloudSim, however most of them turn to focus on comparing just the available algorithms and protocols in CloudSim instead of trying to improve them. In our experimentation, we tried to use our own scheduling algorithm.

We used CloudSim extension CloudAnalyst to simulate Large-scaled applications on the cloud with the purpose of evaluating the behavior of applications under various deployment configurations. This evaluation would benefit developers and system admins immensely in identifying the optional setup for their applications. Additionally, this evaluation will generate valuable insights into designing cloud platform services especially in data centers, load balancing algorithms and potentially optimizing the application performance and cost. Our evaluation is based on three different scenarios of web application hosted on two data centers and four different user bases representing four different regions in Fig. 4 with the simulation setup parameters as can be seen from Fig. 5 and the Tables 2, 3, 4, and 5. For the cost of hosting, we considered the pricing plan of that similar to the one of Amazon EC2 (Figs. 5, 6).

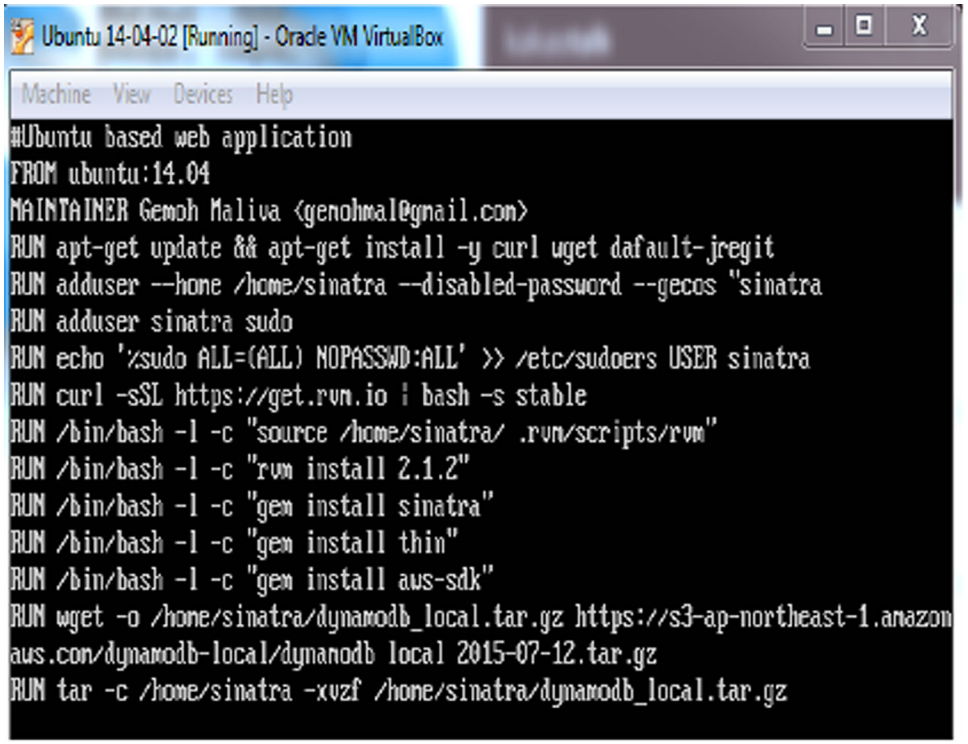

Fig. 5 A demonstration for creating an image using the Dockerfile with parameters

Table 2 User bases setup with region name and configuration

Name	Region	Request per User per Hr	Data Size per Request (bytes)	Peak Hours start (GMT)	Peak Hours End (GMT)	Avg Peak Users	Avg Off-Peak Users
N. America UB	0	60	100	18	20	200,000	20,000
Europe UB	2	60	100	15	17	150,000	15,000
Asia UB	3	60	100	20	22	100,000	10,000
Africa UB	4	60	100	10	12	50,000	5000

Table 3 Application deployment physical configuration in data centers

Data Center	#VMs	Image Size	Memory	BW
DC1	40	100,000	10,240	10,000
DC2	40	100,000	10,240	10,000

Other configurations include the Physical Hardware Details of Data Centers, User grouping factor in User Bases, Request grouping factor in Data Centers, Executable instruction length per request, and the Internet characteristics (Delay Matrix and Bandwidth Matrix).

Experiment results and analysis

Based on the above configurations, the following results were obtained in the three different scenarios. We can clearly see from the graphs during the peak hours how the load changes.

Table 4 Data center cost configuration in data centers

Name	Region	Arch	OS	VMM	Cost Per VM $/Hr	Memory Cost $/Hr	Storage Cost $/Hr	Data Transfer Cost $/Gb	Physical HW Units
DC1	0	x86	Linux	Xen	0.1	0.05	0.1	0.1	30
DC2	2	x86	Linux	Xen	0.1	0.05	0.1	0.1	40

Table 5 Response time and data processing time with total cost in the three scenarios

Method/Scenario	Overall Response Time (ms)	Data Processing Time (ms)	Total Cost VM cost/Data Transfer (dollars)
Scenario 1	168.29	48.16	$168.02/$1138.07
Scenario 2	169.62	38.16	$168.02/$1138.07
Scenario 3	150.49	20.28	$168.02/$1138.07

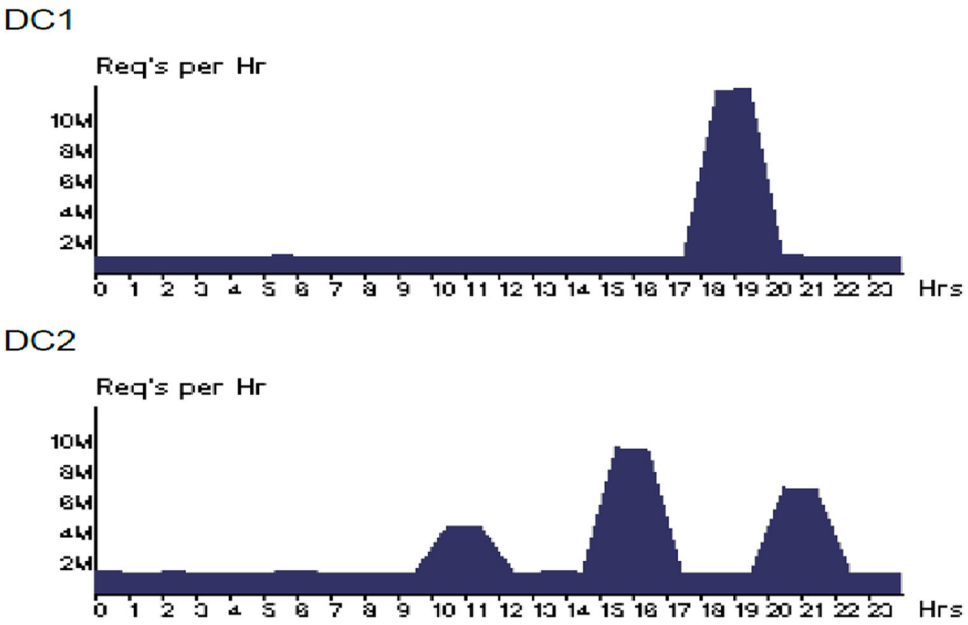

Fig. 6 Data center hourly loading

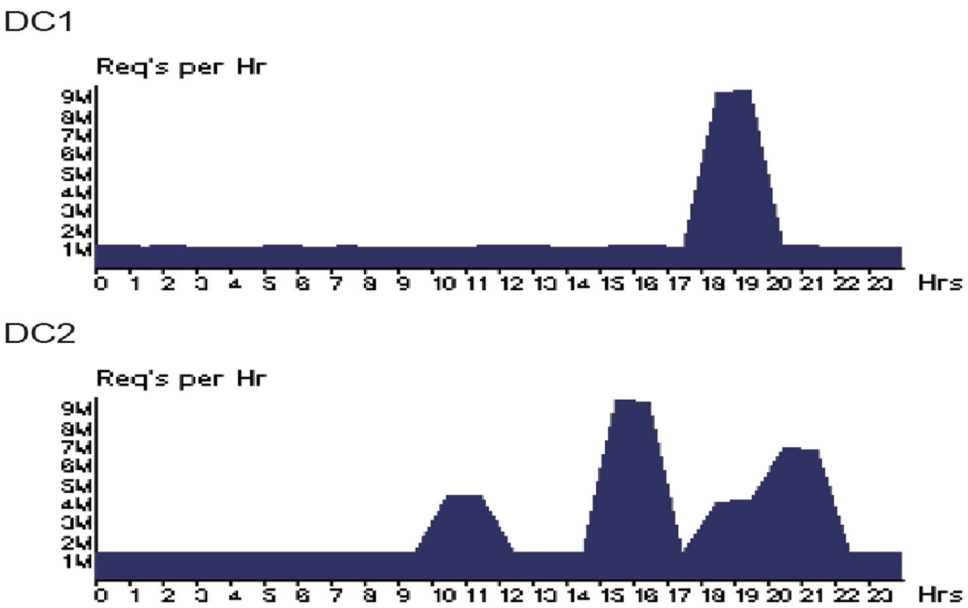

Fig. 7 Data center hourly loading

Scenario 1: Web application hosted on two data centers with 40 VMs in each

Scenario 2: Web application hosted on two data centers with 40 VMs each and sharing the load during peak hours using performance optimize routing.

Scenario 3: Web application hosted on two data centers with 40 VMs each. Applying performance optimized routing and throttled load balancer algorithm.

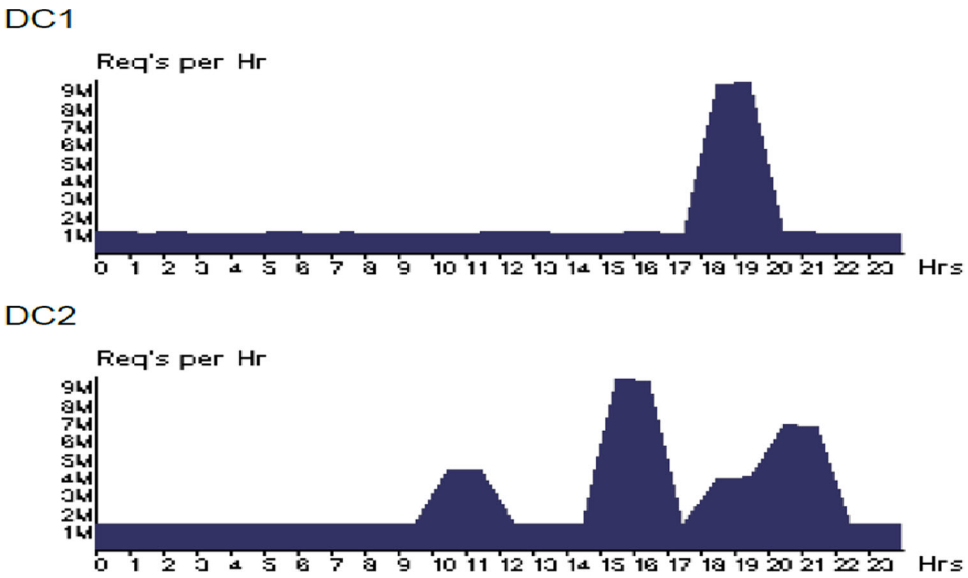

Fig. 8 Data center hourly loading performance of optimized routing and throttled load balancer algorithm

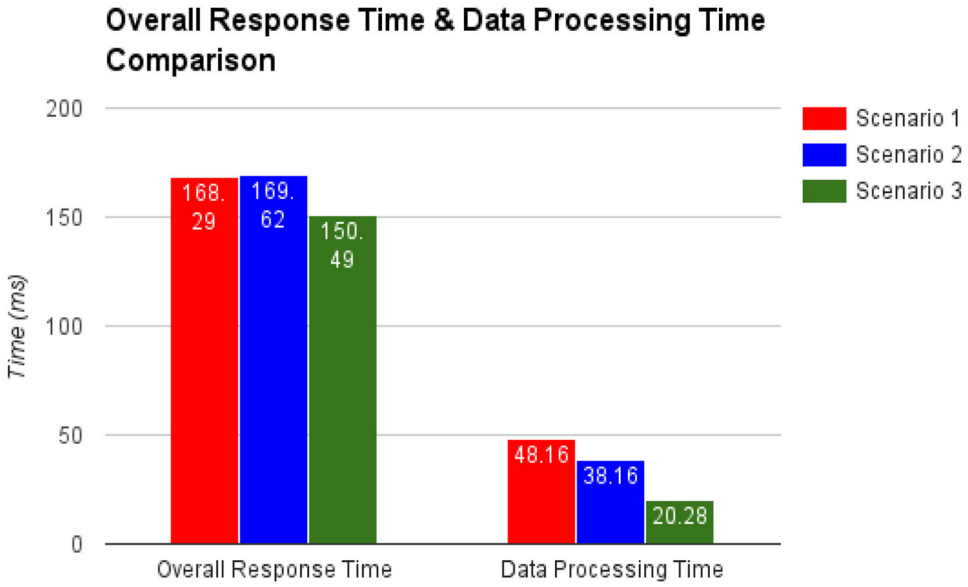

Fig. 9 Data processing time and response time comparison in the three scenarios

Comparing simulation results

Data center hourly loading performance of scenario 1 depicts in Fig. 6, scenario 2 depicts in Fig. 7, and scenario 3 depicts in Fig. 8. The results compare all the three scenarios considering the overall cost and average response time for each of the three scenarios. Due to some unforeseen errors and the limitation of CloudSim at the time of our testing such as CloudSim only supports static assignment with pre-determined resources and tasks, we couldn't apply our proposed schedule algorithm. However, we used the throttled algorithm because is closely similar to our scheduling algorithm although we have added and done improved modification in our algorithm.

From the results of the three scenarios as shown in Fig. 9, undoubtedly we can see that applying performance optimized routing and throttled load balancer algorithm seems to be the best approach for web application hosting and deployment.

6 Conclusion

In this paper, we looked at application optimization and deployment. Based on the challenges in application deployment environment and the numerous advantages of Docker, we proposed a multi-task cloud infrastructure using Docker and AWS services for rapid deployment, application optimization and isolation. We saw that this platform is for

building, shipping, and running applications. We can build any application in any language using any stack, dockerized the application and the application can run anywhere on anything (device). Additionally, we saw how Amazon ECS helps solve challenging problems when running multiple container-based applications and services on a shared compute cluster. Finally we concluded with experiment and evaluations of our work. Our evaluation would benefit developers and system admins immensely in identifying the optional setup for their applications. Moreover, the evaluation generated valuable insights into designing cloud platform services especially in data centers; load balancing algorithms and potentially optimizing the application performance and cost.

For future work, we intend to fully complete the implementation of our proposed cloud platform and scale it up with Amazon EC2 container service for high performance container management. We will then conduct thorough evaluation to demonstrate the flexibility and simplicity of our platform and then compare it with related existing platform.

Acknowledgments This work was supported by 'The Cross-Ministry Giga KOREA Project' grant from the Ministry of Science, ICT and Future Planning, Rep. of Korea (GK16P0100, Development of Tele Experience Service SW Platform based on Giga Media).

References

1. Zhang, Qi, Cheng, Lu, Boutaba, Raouf: Cloud computing: state-of-the-art and research challenges. J. Internet Serv. Appl. **1**(1), 7–18 (2010)
2. Mell, P., Grance, T.: The NIST definition of cloud computing. NIST Special Publication 800–145, Technical Report, pp. 20–23 (2011)
3. Yang, T.A., Joshy, N., Rojas, E., Anumula, S., Moola, J.: Virtualization and data center design. Glob. J. Technol. **9**, 36–54 (2015)
4. Kratzke, N.: Cloud Computing Costs and Benefits. Cloud Computing and Services Science, pp. 185–203. Springer, New York (2012)
5. Kratzke, N.: Lightweight virtualization cluster how to overcome cloud vendor lock-in. J. Comput. Commun. **2**(12), 1–7 (2014)
6. Caballer, M., Blanquer, I., Molto, G., de Alfonso, C.: Dynamic management of virtual infrastructures. J. Grid Comput. **13**(1), 53–70 (2015)
7. Merkel, D.: Docker: lightweight linux containers for consistent development and deployment. Linux J. **239**, 2014 (2014)
8. Nurmi, D., Wolski, R., Grzegorczyk, C., Obertelli, G., Soman, S., Youseff, L., Zagorodnov, D.: The eucalyptus open-source cloud-computing system. In: CCGRID'09. 9th IEEE/ACM International Symposium on Cluster Computing and the Grid, pp. 124–131. IEEE (2009)
9. Caballer, M., Blanquer, I., Molto, G., de Alfonso, C.: Dynamic management of virtual infrastruc-tures. J. Grid Comput. **13**(1), 53–70 (2014)
10. Regola, N., Ducom, J.-C.: Recommendations for virtualization technologies in high performance computing. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp. 409–416. IEEE (2010)
11. Marshall, P., Keahey, K., Freeman, T.: Elastic site: using clouds to elastically extend site resources. In: Proceedings of the 2010 10th IEEE/ACM International Conference on Cluster, Cloud and Grid Computing, pp. 43–52. IEEE Computer Society (2010)
12. LXC - Linux Containers. https://linuxcontainers.org/lxc/introduction (2014)
13. Docker. www.docker.com (2013)
14. Carrion, J.V., Molto, G., De Alfonso, C., Caballer, M., Hernandez, V.: A generic catalog and repository service for virtual machine images. In: 2nd International ICST Conference on Cloud Computing (CloudComp 2010). pp. 1–15 (2010)
15. AmazonWebServices.AWSEC2. http://docs.aws.amazon.com/AmazonECS/latest/developerguide (2014)
16. Keahey, K., Freeman, T.: Contextualization: providing one-click virtual clusters. In: IEEE Fourth International Conference on IEEE eScience eScience'08, pp. 301–308 (2008)
17. Marshall, P., et al.: Architecting a Large-scale Elastic Environment-Recontextualization and Adaptive Cloud Services for Scientific Computing. In: ICSOFT, pp. 409–418 (2012)
18. Bresnahan, J., Freeman, T., LaBissoniere, D., Keahey, K.: Managing appliance launches in infrastructure clouds. In: Proceedings of the 2011 TeraGrid Conference: Extreme Digital Discovery, vol. 12, pp. 1–7. ACM (2011)
19. Binz, T., Breitenbcher, U., Haupt, F., Kopp, O., Leymann, F., Nowak, A., Wagner, S.: OpenTOSCA? A Runtime for TOSCA-Based Cloud Applications. Service-Oriented Computing, pp. 692–695. Springer, Berlin (2013)
20. Papadopoulos, P.M., Katz, M.J., Bruno, G.: NPACI rocks: tools and techniques for easily deploying manageable linux clusters. Concurr. Comput. **00**, 1–20 (2001)
21. Mehra, P.: Guest editor's introduction. IEEE Internet Comput. **5**, 38–40 (2002)
22. Guo, T., Sharma, U., Shenoy, P., Wood, T., Sahu, S.: Cost-aware cloud bursting for enterprise applications. ACM Trans. Internet Technol. (TOIT) **13**(3), 1–22 (2014)
23. Calheiros, R.N., Ranjan, R., Beloglazov, A., De Rose, C.A.F., Buyya, R.: CloudSim: a toolkit for modeling and simulation of cloud computing environments and evaluation of resource provisioning algorithms. J. Softw.-Pract. Exp. **41**(1), 23–50 (2011)

Gemoh Maliva Tihfon received his B.Sc. Degree in Computer Science from Osmania University, Hyderabad-India in 2013. He is currently studying for his Master Degree (M.S.) in the School of Electronics and Computer Engineering, Chonnam National University, Gwangju, South Korea. His research interests include Cloud computing designs and optimization, Cloud virtualization, IoT technology, mobile computing, and other Computer Network related topics.

Sanghyun Park received the B.S. Degree in Computer and Information from the University of Korea Nazarene in 2010, and the M.S. degree in School of Electronics and Computer Engineering, Chonnam National University, South Korea. He worked as an engineer in System Development Team of Media Flow Company from 2010 to 2012. He is now studying Ph.D. Degree in School of Electronics and Computer Engineering, Chonnam National University. His research interests are Interactive Media, Systems Development, Embedded systems, Digital Media and Cloud computing.

Jinsul Kim received the B.S. Degree in computer science from University of Utah, Salt Lake City, Utah, USA, in 2001, and the M.S. and Ph.D. degrees in digital media engineering, department of information and communications from Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea, in 2005 and 2008. He worked as a researcher in IPTV Infrastructure Technology Research Laboratory, Broadcasting/Telecommunications Convergence Research Division, Electronics and Telecommunications Research Institute (ETRI), Daejeon, Korea from 2005 to 2008. He worked as a professor in Korea Nazarene University, Chon-an, Korea from 2009 to 2011. Currently, he is a professor in Chonnam National University, Gwangju, Korea. He has been invited reviewer for IEEE Trans. Multimedia since 2008. He was General Chair of IWICT2013/2014/2015, ICITCS2014, ICISA2015, ICMWT2015 and ICISS2015. His research interests include QoS/QoE, Measurement/Management, IPTV, Mobile IPTV, Smart TV, Multimedia Communication and Smart Space/Works.

Yong-Min Kim received the B.S., M.S. and Ph.D. degrees in Computer Science and Statistics from Chonnam National University in Korea. From 2004 to 2006, he worked at Yeosu National University as a full-time professor. Since 2006, he has been working at Chonnam National University, Yeosu in Korea as an associate professor in Electronic Commerce, Division of Culture Contents. His current research interests include IT Media Convergence, Multimedia Communication, Electronic Commerce System, Internet Security & Privacy.