A Component Framework for Adapting to Elastic Resources in Clouds

Ichiro Satoh
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430, Japan
ichiro@nii.ac.jp

***Abstract*—The notion of *elasticity*, which enables capabilities and resources to be dynamically provisioned and released, is an adaptive mechanism for managing resources in cloud computing. However, most existing applications for cloud computing cannot support *elastic* capabilities and resources. To solve this problem, this paper proposes an approach for adapting distributed applications in response to elastic changes in their resource availability. The approach can divide a component into more than one components and merge more than one components whose program codes are common into a component by using user defined functions for dividing and merging the data stored at key-value stores. It was constructed as a middleware system for general-purpose software components with the two functions. This paper presents the basic ideas, design, and implementation of the approach evaluates the proposed approach.**

Keywords: Cloud computing, Elasticity, Software deployment

I. Introduction

Cloud computing has recently emerged as a compelling paradigm for managing and delivering services over the Internet. The notion of *elasticity*, which enables capabilities and resources to be dynamically provisioned and released, is an adaptive mechanism for managing resources in cloud computing as a material property with the capability of returning to its original state after a deformation. For example, the NIST definition of cloud computing [10] states that capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward in accordance with demand. To the consumer, the capabilities available for provisioning often appear to be unlimited and can be appropriated in any quantity at any time.

In a cloud computing platform, services are delivered with transparency not considering the physical implementation within the platform. However, the conventional design and development of applications for cloud computing are not able to adapt themselves to elastically provisioning and deprovisioning resources in cloud computing. Furthermore, it is difficult to deprive parts of the computational resources that such applications have already used. There have been a few attempts to solve this problem. For example, Mesos [4] is a platform for sharing commodity clusters between distributed data processing frameworks such as Hadoop and Spark. These frameworks themselves are elastic in the sense that they have the ability to scale their resources up or down, i.e., they can start using resources as soon as applications want to acquire the resources or release the resources as soon as the applications do not need them.

This paper assumes that applications are running on dynamic distributed systems, including cloud computing platforms, in the sense that computational resources available from the applications may be dynamically changed due to elasticity. We propose a framework for enabling distributed applications to be adapted to changes in their available resources on elastic distributed systems as much as possible. The key ideas behind the framework are the duplication and migration of running software components and the integration of multiple same components into single components by using the notion of the MapReduce processing [2]. To adapt distributed applications, which consist of software components, to elasticity in cloud computing, the framework divides applications into some of the components and deploys the components at servers, which are provisioned, and merges the components running at servers, which are deprovisioned, into other components running at other available servers. We construct a middleware system for adapting general-purpose software components to changes at elastic resources in cloud computing.

This paper consists of the following sections. In Section II, we surveys related work. Section III present the basic ideas of the approach presented in this paper. Section IV describes the design and implementation of the system. We show the systems' evaluation in Section V and give some concluding remarks Section VI.

II. Related Work

Before presenting our framework, we discuss existing dynamic resource managements in cloud computing, including elastic resource allocation. Cloud computing platforms allow for novel ways of efficient execution and management of complex distributed systems, such as elastic resource provisioning and global distribution of application components. Resource allocation management has been studied for several decades in various contexts in distributed systems, including cloud computing. We focus here on only the most relevant work in the context of large-scale server clusters and cloud computing in distributed systems. Several recent studies have analyzed cluster traces from Yahoo!, Google,

 ISBN: 978-1-61208-532-6

and Facebook and illustrate the challenges of scale and heterogeneity inherent in these modern data centers and workloads. Mesos [4] splits the resource management and placement functions between a central resource manager and multiple data processing frameworks such as Hadoop and Spark by using an offer-based mechanism. Resource allocation is performed in a central kernel and master-slave architecture with a two-level scheduling system. With Mesos, reclaim of resources is handled for unallocated capacity that is given to a framework. The Google Borg system [11] is an example of a monolithic scheduler that supports both batch jobs and long-running services. It provides a single RPC interface to support both types of workload. Each Borg cluster consists of multiple cells, and it scales by distributing the master functions among multiple processes and using multi-threading. YARN [13] is a Hadoop-centric cluster manager. Each application has a manager that negotiates for the resources it needs with a central resource manager. These systems assume the execution of particular applications, e.g., Hadoop and Spark, or can assign resources to their applications before the applications start. In contrast, our framework enables running applications to adapt themselves to changes in their available resources.

Several academic and commercial projects have explored attempts to create auto-scaling applications. Most of them have used static mechanisms in the sense that they are based on models to be defined and tuned at design time. The variety of available resources with different characteristics and costs, variability and unpredictability of workload conditions, and different effects of various configurations of resource allocations make the problem extremely hard if not impossible to solve algorithmically at design time.

Reconfiguration of software systems at runtime to achieve specific goals has been studied by several researchers. For example, Jaeger et al. [6] introduced the notion of self-organization to an object request broker and a publish / subscribe system. Lymberopoulos et al. [9] proposed a specification for adaptations based on their policy specification, *Ponder* [1], but it was aimed at specifying management and security policies rather than application-specific processing and did not support the mobility of components. Lupu and Sloman [8] described typical conflicts between multiple adaptations based on the Ponder language. Garlan et al. [3] presented a framework called Rainbow that provided a language for specifying self-adaptation. The framework supported adaptive connections between operators of components that might be running on different computers. They intended to adapt coordinations between existing software components to changes in distributed systems, instead of increasing or decreasing the number of components.

Most existing attempts have been aimed at provisioning of resources, e.g., the work of Sharman at al. [12]. Therefore, there have been a few attempts to adapt applications to deprovisioned resources. Nevertheless, they explicitly or implicitly assume that their target applications are initially constructed on the basis of master-slave and redundant architectures. Several academic and commercial systems tried introducing *live-migration* of virtual machines (VMs) into their systems, but they could not merge between applications, because they were running on different VMs. Jung et al.[7] have focused on controllers that take into account the costs of system adaptation actions considering both the applications (e.g., the horizontal scaling) and the infrastructure (e.g., the live migration of virtual machines and virtual machine CPU allocation) concerns. Thus, they differ from most cloud providers, which maintain a separation of concerns, hiding infrastructure-level control decisions from cloud clients.

III. Basic Approach

To use *elastic* resources provided in cloud computing platforms, applications need to adapt themselves to changes in their available resources due to elasticity. To solve this problem, we will propose a framework to adapt applications to the provisioning and deprovisioning of servers, which may be running on physical or virtual machines, and software containers, such as Docker, by providing an additional layer of abstraction and automation of virtualization. Our framework assumes that each application consists of one or more software components that may be running on different computers. It has four requirements.

- *Supports elasticity:* Elasticity allows applications to use more resources when needed and fall back afterwards. Therefore, applications need to be adapted to dynamically increasing and decreasing their available resources.
- *Self-adaptation:* Distributed systems essentially lack a global view due to communication latency between computers. Software components, which may be running on different computers, need to coordinate themselves to support their applications with partial knowledge about other computers.
- *Non-centralized management:* There is no central entity to control and coordinate computers. Our adaptation should be managed without any centralized management so that we can avoid any single points of failures and performance bottlenecks to ensure reliability and scalability.
- *Separation of concerns:* All software components should be defined independently of our adaptation mechanism as much as possible. This will enable developers to concentrate on their own application-specific processing.

There are various applications running on a variety of distributed systems. Therefore, the framework should be implemented as a practical middleware system to support general-purpose applications. We also assume that, before the existence of deprovisioning servers, the target cloud

 ISBN: 978-1-61208-532-6

computing platform can notify servers about the deprovisioning after a certain time. Existing commercial or non-commercial cloud computing platform can be classified into three types: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The framework is intended to be used in the second and third, but as much as possible it does not distinguish between the two.

To adapt applications to changes in their available resources due to elasticity, the framework adapts the applications to dynamically provisioning and de-provisioning resources (Fig. 1).

- *Dynamically provisioning resources* When provisioning servers, if a particular component is busy and the servers can satisfy the requirement of that component, the framework divides the component into two components and deploys one of them at the servers, where the divided components have the same programs but their internal data can be replicated or divided in accordance with application-specific data divisions.
- *Dynamically deprovisioning resources* When deprovisioning servers, components running on the servers are relocated to other servers that can satisfy the requirements of the components. If other components whose programs are the same as the former components co-exist on the latter servers, the framework instructs the deployed components to be merged to the original components.

The first and second adaptations need to deploy components at different computers. Our framework introduces mobile agent technology. When migrating and duplicating components, their internal states stored in their heap areas are transmitted to their destinations and are replicated at their clones.

The framework provides another data store for dividing and merging components. To do this, it introduces two notions: *key-value store* (KVS) and *reduce* functions of the *MapReduce* processing. The KVS offers a range of simple functions for manipulation of unstructured data objects, called *values*, each of which is identified by a unique *key*. Such a KVS is implemented as an array of *key* and *value* pairs. Our framework provides KVSs for components so that each component can maintain its internal state in its KVS. Our KVSs are used to pass the internal data of components to other components and to merge the internal data of components into their unified data. The framework also provides a mechanism to divide and merge components with their internal states stored at KVSs by using *MapReduce* processing. *MapReduce* is a most typical modern computing models for processing large data sets in distributed systems. It was originally studied by Google [2] and inspired by the *map* and *reduce* functions commonly used in parallel list processing (LISP) and functional programming paradigms.

- *Component division* Each duplicated component can inherit partial or all data stored in its original component in accordance with user-defined *partitioning* functions, where each function map of each item of data in its original component's KVS is stored in either the original component's KVS or the duplicated component's KVS without any redundancy.
- *Component fusion* When unifying two components that generated from the same programs into a single component, the data stored in the KVSs of the two components are merged by using user-defined *reduce* functions. These functions are similar to the *reduce* functions of MapReduce processing. Each of our *reduce* functions processes two values of the same keys and then maps the results to the entries of the keys. Figure 1 shows two examples of *reduce* functions. The first concatenates values in the same keys of the KVSs of the two components, and the second sums the values in the same keys of their KVSs.

IV. Design and Implementation

This section presents the design and implementation of our framework. It consists of two parts: component runtime system and components. The former is responsible for executing, duplicating, and migrating components. The later is autonomous programmable entities like software agents. The current implementation is built on our original mobile agent platform as existing mobile agent platforms are not optimized for data processing.

A. Adaptation for elastic resources

When provisioning servers, the framework can divide a component into two components whose data can be divided before deploying one of them at the servers. When deprovisioning servers, the framework can merge components that are running on the servers into other components.

1) Dividing component: When dividing a component into two, the framework has two approaches for sharing between the states of the original and clone components.

- *Sharing data in heap space* Each runtime system makes one or more copies of components. The runtime system can store the states of each agent in heap space in addition to the codes of the agent in a bit-stream formed in Java's JAR file format, which can support digital signatures for authentication. The current system basically uses the Java object serialization package for marshalling agents. The package does not support the capturing of stack frames of threads. Instead, when an agent is duplicated, the runtime system issues events to it to invoke their specified methods, which should be executed before it is duplicated, and it then suspends their active threads.
- *Sharing data in KVS* When dividing a component into two components, the KVS inside the former is

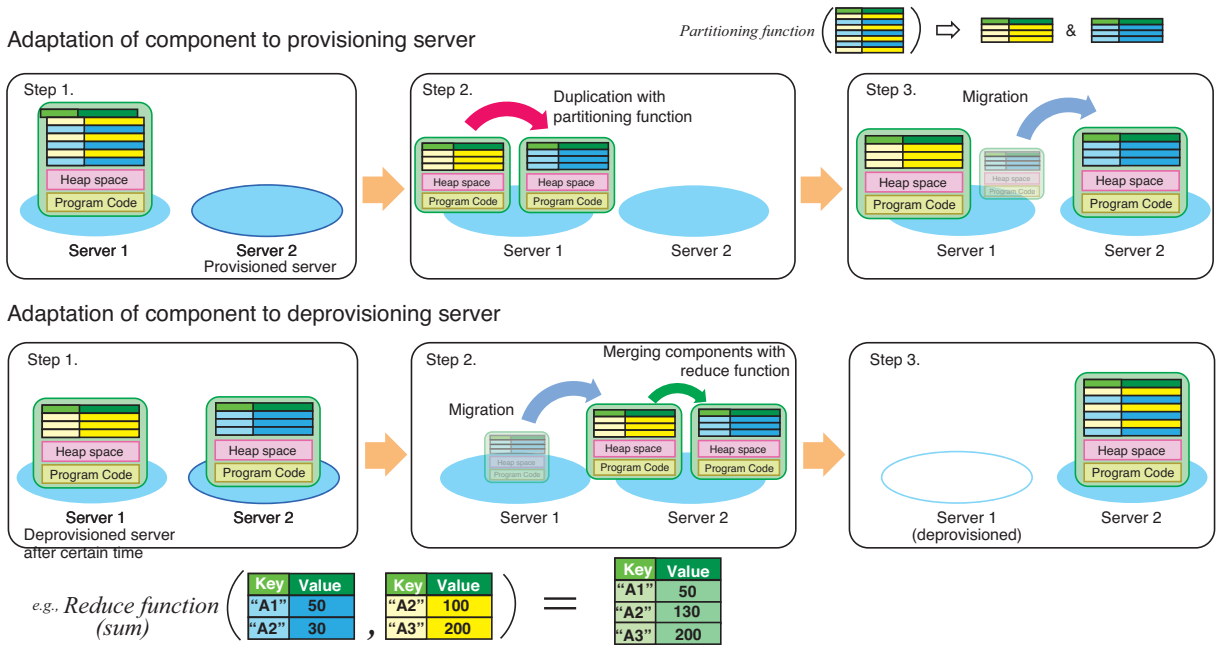

Figure 1. Adaptation to (de)provisioning servers

divided into two KVSs in accordance with user-defined partitioning functions in addition to built-in functions, and the divided KVSs are maintained inside the latter. Partitioning functions are responsible for dividing the intermediate key space and assigning intermediate key-value pairs to the original and duplicated components. In other words, the partition functions specify the components to which an intermediate key-value pair must be copied. KVSs are constructed as in-memory storage to exchange data between components. It provides tree-structured KVSs inside components. In the current implementation, each KVS in each data processing agent is implemented as a hash table whose keys, given as pairs of arbitrary string values, and values are byte array data, and it is carried with its agent between nodes,

where a default partitioning function is provided that uses hashing. This tends to result in fairly well-balanced partitions. The simplest partitioning functions involve computing the hash value of the key and then taking the mod of that value using the number of the original and duplicated components.

2) Merging components: The framework provides a mechanism to merge the data stored in the KVSs of different components instead of the data stored inside their heap spaces. Like the *reduce* of MapReduce processing, the framework enables us to define a *reduce* function that merges all intermediate values associated with the same intermediate key. When merging two components, the framework can discard the states of their heap spaces or keep the state of the heap space of one of them. Instead, the data stored in the KVSs of different components can be shared. A *reduce* function is applied to all values associated with the same intermediate key to generate output key-value pairs. The framework can merge more than two components at the same computers because components can migrate to the computers that execute co-components that the former wants to merge to.

V. Evaluation

Although the current implementation was not constructed for performance, we evaluated the performance of our current implementation. We evaluated the performance of our framework with CoreOS, which is a lightweight operating system based on Linux with JDK version 1.8 with Docker, which is software-based environment that automates the deployment of applications inside software containers by providing an additional layer of abstraction and automation of operating-system-level virtualization on Linux, on Amazon EC2. For each dimension of the adaptation process with respect to a specific resource type, elasticity captures the following core aspects of the adaptation:

- *Adaptation latency at provisioning servers* The response time of scaling up is defined as the time it takes to switch from provisioning of servers by the underlying system, e.g., cloud computing platform.
- *Adaptation latency at deprovisioning servers* The response time of scaling down is defined as the time it takes to switch from deprovisioning of servers by the underlying system, e.g., cloud computing platform.

The latency at scaling up or down does not correspond directly to the technical resource provisioning or deprovi-

sioning time. Table I shows the basic performance. The component was simple and consisted of basic callback methods. The cost included that of invoking two callback methods. The cost of component migration included that of opening TCP transmission, marshaling the agents, migrating the agents from their source computers to their destination computers, unmarshalling the components, and verifying security.

Table I
BASIC OPERATION PERFORMANCE

	Latency (ms)
Duplicating component	10
Merging component	8
Migrating component between two servers	32

Figure 2 shows the latency of the number of divided and merged components at provisioning and deprovisioning servers. The experiment provided only one server to run our target component, which was a simple HTTP server (its size was about 100 KB). It added one server every ten seconds until there were eight servers and then removed one server every ten seconds after 80 seconds had passed. The number of components was measured as the average of the numbers in ten experiments. Although elasticity is always considered with respect to one or more resource types, the experiment presented in this paper focuses on cloud computing platforms for executing components, e.g., servers. There are two metrics in an adaptation to elastic resources, *scalability* and *efficiency*, where scalability is the ability of the system to sustain increasing workloads by making use of additional resources, and efficiency expresses the amount of resources consumed for processing a given amount of work.

- $\overline{A}$ is the average time to switch from an underprovisioned state to an optimal or overprovisioned state and corresponds to the average latency of scaling up or scaling down.
- $\overline{U}$ is the average amount of underprovisioned resources during an underprovisioned period. $\sum \overline{U}$ is the accumulated amount of underprovisioned resources and corresponds to the blue areas in Fig. 2.
- $\overline{D}$ is the average amount of overprovisioned resources during an overprovisioned period. $\sum \overline{D}$ is the accumulated amount of underprovisioned resources and corresponds to the red areas in Fig. 2.

The precision of scaling up or down is defined as the absolute deviation of the current amount of allocated resources from the actual resource provisioning or deprovisioning. We define the average precision of scaling up P_u and that of scaling down P_d. The efficiency of scaling up or down is defined as the absolute deviation of the accumulated amount of underprovisioned or overprovisioned resources from the accumulated amount of provisioned or deprovisioned resources, specified as E_U or E_D.

$$P_u = \frac{\sum \overline{U}}{T_u} \quad P_d = \frac{\sum \overline{D}}{T_d} \quad E_u = \frac{\sum \overline{U}}{R_u} \quad E_d = \frac{\sum \overline{D}}{R_d}$$

where T_u and T_d are the total durations of the evaluation periods and R_u and R_d are the accumulated amounts of provisioned resources when scaling up and scaling down phases, respectively. Table II shows the precision and efficiency of our framework.

Table II
BASIC OPERATION EFFICIENCY

	Rate
P_u (Precision of scaling up)	99.2 %
P_d (Precision of scaling down)	99.1 %
E_u (Efficiency of scaling up)	99.6 %
E_d (Efficiency of scaling down)	99.4 %

In the experiment the target component is a simple HTTP server, since web applications have very dynamic workloads generated by variable numbers of users, and they face sudden peaks in the case of unexpected events. Therefore, dynamic resource allocation is necessary not only to avoid application performance degradation but also to avoid underutilized resources. The experimental results showed that our framework could follow the elastically provisioning and deprovisioning of resources quickly, and the number of the components followed the number of elastic provisioning and deprovisioning of resources exactly. The framework was scalable because its adaptation latency was independent of the number of servers.

VI. CONCLUSION

This paper presented a mechanism for adapting application-level software to changes in available resources in cloud computing platforms. The mechanism was constructed as a framework that enabled distributed applications to adapt themselves to changes in their available resources in distributed systems, in particular cloud computing platforms. It was useful for adapting applications to elasticity in cloud computing. The key ideas behind the framework are *dynamic deployment of components* and *dividing and merging components*. The former enabled components to relocate themselves at new servers when provisioning the servers and at remaining servers when de-provisioning the servers, and the latter enables the states of components to be divided, and passed to other components, and merged with other components in accordance with user-defined functions. We believe that our framework is useful because it enables applications to be operated with elastic capabilities and resources in cloud computing.

REFERENCES

[1] N. Damianou, N. Dulay, E. Lupu, and M. Sloman: The Ponder Policy Specification Language, in Proceedings of Workshop on Policies for Distributed Systems and Networks (POLICY'95), pp.18–39, Springer-Verlag, 1995.

 ISBN: 978-1-61208-532-6

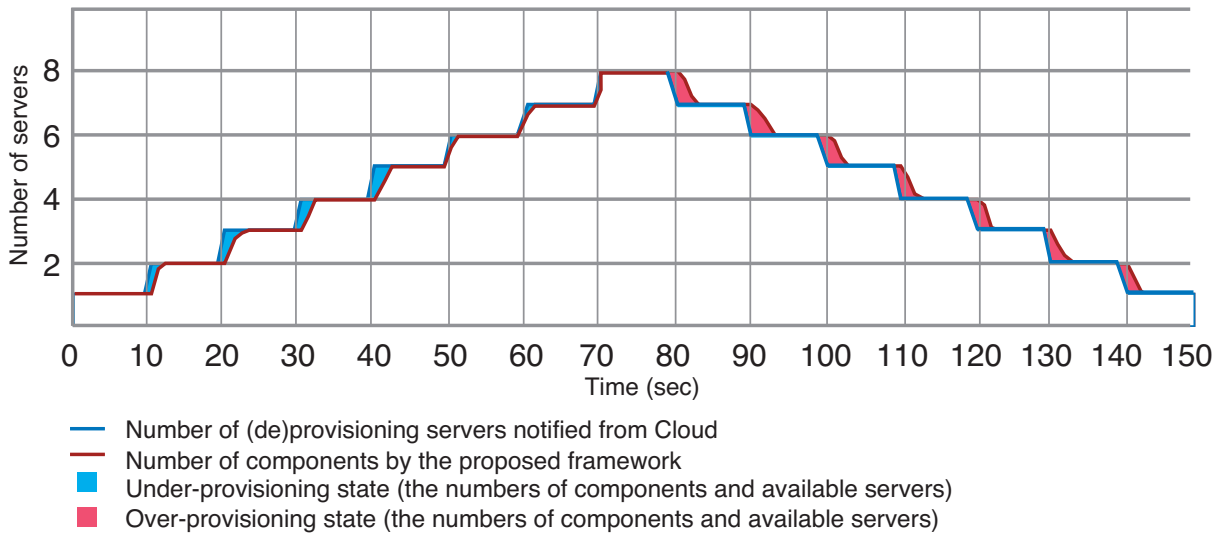

Figure 2. Number of components at (de)provisioning servers

[2] J. Dean and S. Ghemawat: MapReduce: simplified data processing on large clusters, in Proceedings of the 6th conference on Symposium on Opearting Systems Design and Implementation (OSDI'04), 2004.

[3] D. Garlan, S.W. Cheng, A.C.Huang, B. R. Schmerl, P. Steenkiste: Rainbow: Architecture-Based Self-Adaptation with Reusable Infrastructure, IEEE Computer Vol.37, No.10, pp.46-54, 2004.

[4] B. Hindman, A. Konwinski, M. Zaharia, A. Ghodsi, A. Joseph, R. Katz, S. Shenker, and I. Stoica. Mesos: a platform for fine-grained resource sharing in the data center In Proceedings of USENIX Symposium on Networked Systems Design and Implementation (NSDI), 2011.

[5] C. Inzinger, at al., Decisions, Models, and Monitoring–A Lifecycle Model for the Evolution of Service-Based Systems, In Proceedings of Enterprise Distributed Object Computing Conference (EDOC), pp.185-194, IEEE Computer Society, 2013.

[6] M. A. Jaeger, H. Parzyjegla, G. Muhl, K. Herrmann: Self-organizing broker topologies for publish/subscribe systems, in Proceedings of ACM symposium on Applied Computing (SAC'2007), pp.543-550, ACM, 2007.

[7] G. Jung, et. al.: A Cost-Sensitive Adaptation Engine for Server Consolidation of Multitier Applications, In Proceedings of Middleware'2009, LNCS, Vol.5896, pp.163183, Springer, 2009.

[8] E. Lupu and M. Sloman: Conflicts in Policy-Based Distributed Systems Management, IEEE Transaction on Software Engineering, Vol.25, No.6, pp.852-869, 1999.

[9] L. Lymberopoulos, E. Lupu, M. Sloman: An Adaptive Policy Based Management Framework for Differentiated Services Networks, in Proceedings of 3rd International Workshop on Policies for Distributed Systems and Networks (POLICY 2002), pp.147-158, IEEE Computer Society, 2002.

[10] P. Mell, T. Grance: The NIST Definition of Cloud Computing, Technical report of U.S. National Institute of Standards and Technology (NIST), Special Publication 800-145, 2011.

[11] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes: Large-scale cluster management at Google with Borg, EuroSys15, ACM 2015.

[12] U. Sharma, P. Shenoy, S. Sahu, A. Shaikh: A cost-aware elasticity provisioning system for the cloud In Proceedings of International Conference on Distributed Computing Systems (ICDCS'2011), pp.559570, IEEE Computer Society, 2011.

[13] V. K. Vavilapalli, el. al.,: Apache Hadoop YARN: Yet Another Resource Negotiator, In Proceedings of Symposium on Cloud Computing (SoCC'2013), ACM, 2013.

[14] World Wide Web Consortium (W3C): Composite Capability/Preference Profiles (CC/PP), http://www.w3.org/TR/NOTE-CCPP, 1999.

 ISBN: 978-1-61208-532-6