Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 94 (2016) 9 – 16

Procedia Computer Science

The 13th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2016)

Self-Adaptively Auto-scaling for Mobile Cloud Applications

Ichiro Satoh

National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo 101-8430 Japan

Abstract

This paper proposes an approach to adapting distributed applications to changes in user requirements and resource availability. The key ideas behind the framework were *dynamic deployment of components* and *dividing and merging components*. The former enabled components to relocate themselves at new servers when provisioning the servers and remained servers when deprovisioning servers. The latter enabled the states of components to be divided and passed to other components and to be merged with other components according to user-defined functions. It was useful to adapt applications to *elasticity* in cloud computing. It is constructed as a middleware system for Java-based general-purposed software components. This paper describes the proposed approach and the design and implementation of the approach with applications.

Peer-review under responsibility of the Conference Program Chairs

Keywords: Context-aware services; User evaluation; User tracking

1. Introduction

Mobile computing is characterized by many constraints: non-powerful processors, small memory and storage, and battery-powered portable devices. These are not merely a temporary technological deficiency but intrinsic to mobility, and a barrier that needs to be overcome in order to realize the full potential of mobile computing. On the other hand, modern applications in mobile computers with such constraints are required to provide more enriched and sophisticated functions than ever. Therefore, such applications cannot be longer constructed as standalone ones. Instead, applications running on mobile computing often access information from servers available in their current networks and delegate heavy tasks to the servers. Cloud computing has emerged as one of the most important computing strategies in modern enterprise systems. Cloud computing can provides a variety of computational resources, including servers, for mobile computers. The concept of offloading data and computation in cloud computing, is used to address the inherent problems in mobile by using resource providers other than the mobile computer itself to host the execution of mobile applications.

Cloud computing environments allow for novel ways of efficient execution and management of complex distributed systems, such as elastic resource provisioning and global distribution of application components. *Elasticity* was orig-

* Corresponding author. Tel.: +8-3-4212-2546.
E-mail address: author@institute.xxx

Peer-review under responsibility of the Conference Program Chairs
doi:10.1016/j.procs.2016.08.006

inally defined in physics as a material property capturing the capability of returning to its original state after a deformation. The concept of elasticity has been transferred to the context of cloud computing and is commonly considered one of the central attributes of cloud computing. For example, the NIST Definition of cloud computing[9] states that capabilities can be elastically provisioned and released, in some cases automatically, to scale rapidly outward and inward commensurate 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. However, conventional application design and development are not able to adapt themselves to elastic resource provisioning in cloud computing. Furthermore, it is difficult to deprive parts of the computational resources that such applications have already used.

In this paper we assume that applications are running on dynamic distributed systems, including cloud computing environments, 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. To adapt distributed applications, which consist of software components, to elasticity in cloud computing, the framework divides some of the components and deploys them at servers, which are provisioned, and merges components running at servers, which are de-provisioned, into other components running at other available servers. We are constructing a middleware system for adapting general-purpose software components to changes in elastic cloud computing.

2. Related Work

Resource allocation management has been studied for several decades in contexts as varied as distributed systems. 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, and Facebook and illustrate the challenges of scale and heterogeneity inherent in these modern data-centers and workloads. Mesos[3] 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 done in a central kernel and master-slave architecture with a two-level scheduling system. In Mesos, reclaim of resources is handled for unallocated capacity that is given to a framework. The Google'Borg system[10] 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 to execute 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.

There have been many 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. For example, Tamura et al.[12] proposed an approach to identify system viability zones that are defined as states in which the system operation is not compromised and to verify whether the current available resources can satisfy the validation at the development of the applications. The variety of available resources with different characteristics and costs, the variability and unpredictability of workload conditions, and the 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 for achieving specific goals has been studied by several researchers. For example, Jaeger et al.[5] introduced the notion of self-organization to an object request broker and a publish / subscribe system. Lymberopoulos et al.[8] 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[7] described typical conflicts between multiple adaptations based on the Ponder language. Garlan et al.[2] presented a framework called Rainbow that provided a language for specifying self-adaptation. Although the framework was not aimed at only distributed systems, it 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 components.

Most existing attempts have been aimed at provisioning of resources, e.g.,[11]. Therefore, there have been a few attempts to adapt applications to de-provisioned 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, which were running on different VMs.[1] Jung et al.[6] 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 that maintain a separation of concerns, hiding infrastructural control decisions from cloud clients.

3. Approach

This section briefly outlines our framework. As mentioned in the first section, *elasticity*, which is one of the most important features of cloud computing, is the degree to which a system is able to adapt to workload changes by provisioning and de-provisioning resources in an autonomic manner. Applications need to adapt themselves to changes in their available resources due to elasticity.

3.1. Requirements

In this paper our aim is to adapt applications to the provisioning and de-provisioning of servers, which may be running on physical or virtual machines, and software containers, 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. 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. All software components should be defined independently of our adaptation mechanism as much as possible for separation of concerns. This will enable developers to concentrate on their own application-specific processing. 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 bottleneck to ensure reliability and scalability. We assume that, before de-provisioning servers, the target cloud computing environment can notify servers about the de-provisioning after a certain time. Cloud computing environments can be classified into three types: Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS). The framework intend to be used in the second and third, but as much as possible does not distinguish between the two.

3.2. Adaptation for elasticity in cloud computing

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

- *Adaptation for 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.
- *Adaptation for de-provisioning resources* When de-provisioning servers, components running on the servers are relocated at 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 instruct the deployed components to be merged to the original components.

[1] Unlike private cloud cloud environments, most commercial ones do not support live migrations because such live migrations tend to need wide-band and low-latency networks between servers.

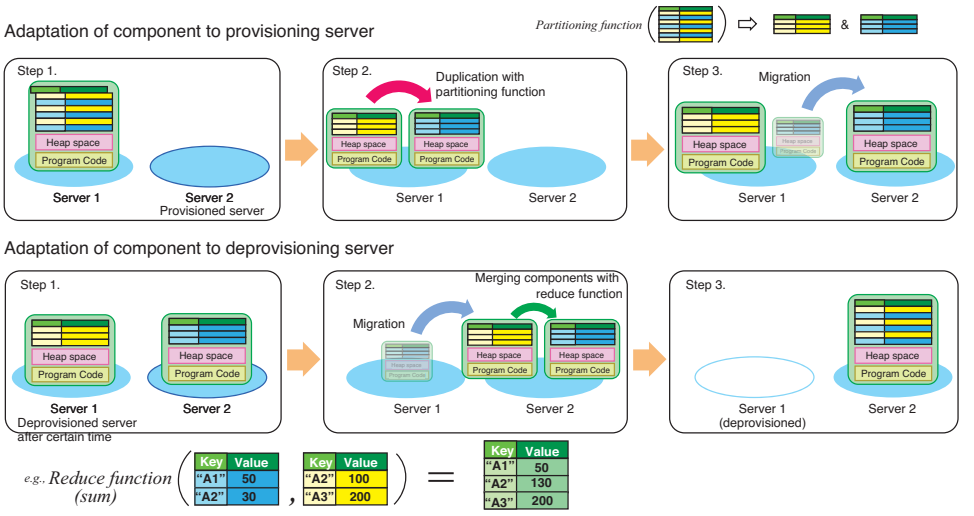

Fig. 1. Adaptation to (de)provisioning servers

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 area are transmitted to their destinations and are replicated at their clones.

3.3. Data stores for dividing and merging components

The framework also provides another data store for dividing and merging components. To do this, it introduces two notions: a *key-value store* (KVS) and a *reduce* function 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* paris. 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 one of the most typical modern computing models for processing large data sets in distributed systems.

- *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 stored in either the original component's KVS or the duplicated component's KVS without any redundancy.
- *Component fusion* When unifying two same components into a single component, the data stored in the KVSs of 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 two components and the second sums the values in the same keys of their KVSs.

4. Implementation

This section describes our component runtime system and components. The system is responsible for executing, duplicating, and migrating components. These components are 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.

4.1. Component

Each component is an autonomous programmable entity. We can define each component as a collection of Java objects and a KVS. Each method can be invoked from other components, which may be running on different computers via our original remote method invocation (RMI) mechanism, which can be automatically handled in the mobility and duplication of components. One-way message transmission, publish-subscription events, and stream communications are supported by using the RMI mechanisms. The current implementation uses the standard JAR file format for archiving components because this format can support digital signatures, enabling authentication. Each agent can specify the requirements that its destination hosts must satisfy in CC/PP form [14] and the runtime system can select an appropriate destination among multiple destination candidates through comparing between the capabilities required by agents and the capabilities of the candidates. When an agent is deployed at another computer, the runtime system invokes a specified callback method defined in the annotation part and then one defined in the navigation part. Although these parts are implemented as Java objects, they are loosely connected with one another through data attributes by using Java's introspection mechanism so that they can be replaced without any compilations or linkages for their programs.

4.2. Runtime system

Each component can have one or more activities, which are implemented by using the Java thread library. Furthermore, the runtime system maintains the lifecycles of components. When the life-cycle state of an component is changed, the runtime system issues certain events to the component. The system can impose specified time constraints on all method invocations between components to avoid being blocked forever. Each component is provided with its own Java class load, so that its namespace is independent of other components in each runtime system. The identifier of each component is generated from information consisting of its runtime system's host address and port number, so that each component has a unique identifier in the whole distributed system. Therefore, even when two components are defined from different classes whose names are the same, the runtime system disallows components from loading other components's classes. To prevent components from accessing the underlying system and other components, the runtime system can control all components under the protection of Java's security manager.

Each runtime system also establishes at most one TCP connection with each of its neighboring systems in a peer-to-peer manner without any centralized management server and it exchanges control messages and agents through the connection. When an agent is transferred over a network, the runtime system transfers the agent into a bit-stream like task duplication and transmits the bit-stream to the destination data nodes through TCP connections from the source node to the nodes. After they arrive at the nodes, they are resumed and activated from the marshalled agents and then their specified methods are invoked to acquire resources and continue processing. When provisioning or de-provisioning servers, the underlying cloud computing environment notifies about such changes to the servers and neighboring servers.

4.3. Adaptation for elasticity

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 de-provisioning servers, the framework can merge components that are running on the servers into other components.

4.3.1. 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 marshaling 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 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 components. Partitioning functions are responsible for dividing up 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 involves computing the hash value of the key and then taking the mod of that value with the number of the original and duplicated components.

4.3.2. 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.

5. Evaluation

A prototype implementation of this framework was constructed with Sun's Java Developer Kit (JDK) version 1.7 or later versions. The implementation provided graphical user interfaces to operate the mobile agents. Although the current implementation was not constructed for performance, we evaluated the performance of our framework with CoreOS, which is a lightweight operating system based on Linux with JDK version 1.8 in Docker, which is an 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 speed at provisioning servers* The speed of scaling up is defined as the time it takes to switch from provisioning of servers by the underlying system, e.g., cloud computing environment.
- *Adaptation speed at de-provisioning servers* The speed of scaling down is defined as the time it takes to switch from de-provisioning of servers by the underlying system, e.g., cloud computing environment.

The speed of scaling up/down does not correspond directly to the technical resource provisioning/de-provisioning time. Table 1 shows the basic performance. The component was simple and consisted of basic callback methods. The cost included that of invocating 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, unmarshaling the components, and verifying security.

Figure 2 shows the speed of the number of divided and merged components at provisioning and de-provisioning servers. The experiment provided only one server to 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

Table 1. Basic operation performance

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

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 computing environments 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.

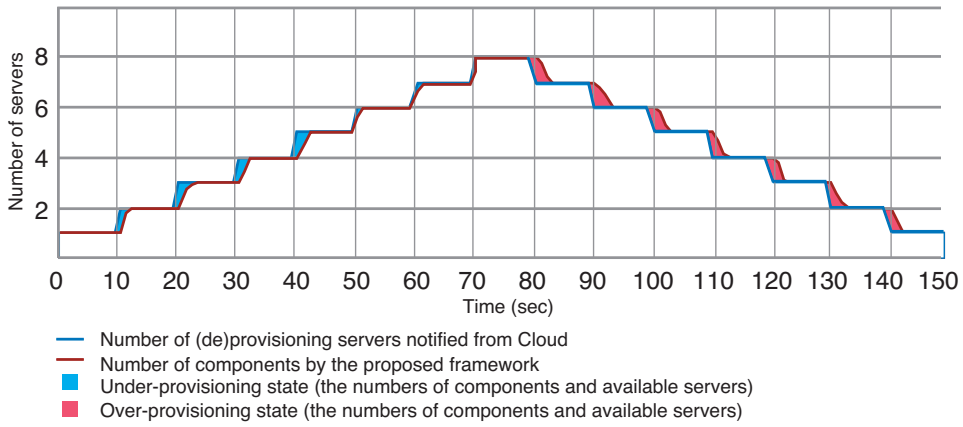

Fig. 2. Number of components at (de)provisioning servers

- $\overline{A}$ is the average time to switch from an underprovisioned state to an optimal or overprovisioned state and corresponds to the average speed 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 de-provisioning. 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 de-provisioned resources, specified as E_U or E_D, where $P_u = \frac{\sum \overline{U}}{T_u}$, $P_d = \frac{\sum \overline{D}}{T_d}$ $E_u = \frac{\sum \overline{U}}{R_u}$, $E_d = \frac{\sum \overline{D}}{R_d}$, 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.[2] Table 2 shows the precision and efficiency of our framework.

Table 2. 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 %

[2] R_u and R_d correspond to the amount of provisioned resources notified from cloud computing environments.

Our component corresponds to an 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 under-utilized resources. The experimental results showed that our framework could follow the elastically provisioning and de-provisioning of resources quickly and the number of the components followed the number of elastic provisioning and de-provisioning of resources exactly. The framework was scalable because its adaptation speed was independent of the number of servers.

6. Conclusion

We presented a framework for enabling distributed applications to be adapted to changes in their available resources in distributed systems. 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 remained servers when de-provisioning the servers and the latter enables the states of components to be divided and passed to other components and to be merged with other components in accordance with user-defined functions.

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.
2. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. 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.
8. 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.
9. 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.
10. A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes: Large-scale cluster management at Google with Borg, EuroSys15, ACM 2015.
11. 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.
12. G. Tamura et. al.,: Towards Practical Runtime Verification and Validation of Self-Adaptive Software Systems, Proceedings of Self-Adaptive Systems, LNCS 7475, pp. 108132, 2013.
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.