A Kubernetes-Based Monitoring Platform for Dynamic Cloud Resource Provisioning

Chia-Chen Chang and Shun-Ren Yang
Dept. of Comp. Sci.
National Tsing Hua University
Hsinchu, Taiwan, R.O.C.
sryang@cs.nthu.edu.tw

En-Hau Yeh and Phone Lin
Dept. of Comp. Sci.
National Taiwan University
Taipei, Taiwan, R.O.C.
plin@csie.ntu.edu.tw

Jeu-Yih Jeng
Billing Information Laboratory
Telecommunication Laboratories
Chunghwa Telecom Co., Ltd.,
Taipei, Taiwan, R.O.C.
jyjeng@cht.com.tw

***Abstract*—Recently, more and more network operators have deployed cloud environment to implement network operations centers that monitor the status of their large-scale mobile or wireline networks. Typically, the cloud environment adopts container-based virtualization that uses Docker for container packaging with Kubernetes for multihost Docker container management. In such a container-based environment, it is important that the Kubernetes can dynamically monitor the resource requirements and/or usage of the running applications, and then adjust the resource provisioned to the managed containers accordingly. Currently, Kubernetes provides a naive dynamic resource-provisioning mechanism which only considers CPU utilization and thus is not effective. This paper aims at developing a generic platform to facilitate dynamic resource-provisioning based on Kubernetes. Our platform contains the following three features. First, our platform includes a comprehensive monitoring mechanism that integrates and provides the relatively complete system resource utilization and application QoS metrics to the resource-provisioning algorithm to make the better provisioning strategy. Second, our platform modularizes the operation of dynamic resource-provisioning operation so that the users can easily deploy a newly designed algorithm to replace an existing one in our platform. Third, the dynamic resource-provisioning operation in our platform is implemented as a control loop which can consequently be applied to all the running application following a user-defined time interval without other manual configuration.**

***Index Terms*—Container, Docker, dynamic resource provisioning, monitoring, Kubernetes.**

I. Introduction

Recently, more and more network operators (e.g., Chunghwa Telcom, Taiwan) have deployed and operated large data centers to manage their businesses. Specifically, the operators have adopted cloud technologies to implement network operations centers that monitor the status of their large-scale mobile or wireline networks, such as computing-resource usage and network response time.

In data centers, container-based virtualization, also known as operating system (OS)-level virtualization, is a new virtualization technology that divides a server into multiple virtual instances, called containers, by sharing the underlying OS kernel. Unlike traditional virtual machines (VMs), lightweight containers can be started rapidly, which brings about a lower overhead of computing resources and time in deployment.

One of the most popular solutions for container-based virtualization is using Docker for container packaging with Kubernetes for multihost container management [1]. Kubernetes follows the masterslave model, which uses a master to manage Docker containers across multiple Kubernetes nodes (which are physical or virtual machines). A master and its controlled nodes constitute a "Kubernetes cluster." A developer can deploy an application in the docker containers via the assistance of the Kubernetes master. Typically, an application is divided into one or more tasks executed in one or more containers. Each container is generally restricted in terms of the maximum resource quantity that it can consume. That is to say, the total amount of resource provisioned to an application is the sum of its executed containers' resource capacity.

As mentioned above, Kubernetes is a tool which is used to manage multiple Docker containers across multiple machines. In such a container-based environment, it is important that the Kubernetes can dynamically monitor the resource requirements and/or usage of the running applications, and then adjust the resource provisioned to the managed containers accordingly, preventing from resource overprovisioning and underprovisioning. Typically, the dynamic resource provisioning procedure can be simply separated into four steps: (1) *The monitor step:* Monitoring the resource usage and limitation of the entire environment and all of the running applications; (2) *The analyze step:* Integrating the monitored data as the input data of the plan step; (3) *The plan step:* Using an resource provisioning algorithm to determine the resource-provisioning strategy based on the data generated by the analyze step; and (4) *The execute step:* Adjusting the resource provisioning to all of the running applications according to the strategy in the plan step.

In Kubernetes, a built-in mechanism has been provided for dynamic resource provisioning. In this mechanism, the users can first set a desired percentage of CPU utilization to a specific group of containers that are executed for an application. Then, the mechanism executes the four steps that implement dynamic resource provisioning. In particular, in the monitor and analyze steps, Kubernetes uses a user-deployed monitoring tool to retrieve the CPU utilization of each running container; in the plan step, Kubernetes uses its own algorithm to determine the number of containers to ensure that the average CPU utilization approaches the user's desired quantity. However, this mechanism has some deficiencies. In the plan step, Kubernetes only allows users execute its own resource-provisioning algorithm; in other words, users cannot apply another resource-provisioning algorithm by themselves. This suggests that the mechanism is not flexible. Furthermore, the algorithm only considers CPU utilization when determining the resource provisioning strategy. This is not realistic enough because there are many factors affecting the application performance. Lastly, this mechanism has to be invoked manually for each application. Therefore, the more running applications, the more redundant manual configurations that users have to set.

This paper develops a generic platform to facilitate dynamic resource provisioning over Kubernetes, while resolving the existing mechanism's deficiencies. Specifically, our platform has the following features:

- *A comprehensive monitoring mechanism.* Unlike the existing Kubernetes built-in mechanism that only considers CPU utilization, our platform takes both system resource utilization and application QoS metrics into account. Specifically, our platform employs an open-source tool to retrieve all the resource utilization metrics (such as memory and disk access) that the underlying system can provide. Moreover, we integrate the application QoS metrics (such as response time) monitoring into our platform. Thus, a rather complete set of monitored metrics can be exposed to the resource-provisioning algorithm, for better provisioning decisions;
- *Deployment flexibility.* In our platform, we modularize the four steps of dynamic resource-provisioning, so that the users can easily deploy a newly designed algorithm to replace an existing one without affecting other modules;
- *Automatic operation.* After the users deploy a desired dynamic resource-provisioning algorithm, the resource-provisioning operation will be applied automatically and periodically to all the running applications in our platform following a user-defined time interval.

The remainder of the paper is organized as follows: Section 2 gives a brief introduction to Docker. Section 3 describes the system architecture and components of our platform. Section 4 illustrates the implementation of dynamic-resource provisioning using our platform. Section 5 describes one experiment performed to verify the operations of dynamic resource provisioning in our platform. Finally, the paper is concluded in Section 6.

II. Overview of Docker Container-based Virtualization

There have been a lot of implementations of container-based virtualization, such as LXC, lmctfy, FreeBSD jail, and Docker. In particular, Docker [2], an open source project that was initially released in 2013, has drawn great attention from the IT industry. Docker is a software containerization platform that allows users to deploy their application inside a container and transfer containers across machines in a simple way. There are three main components of Docker, as follows:

- *Docker images.* Docker images are read-only templates which are the bases of creating Docker containers.;
- *Docker registries.* Docker registries are the storage that holds a huge collection of Docker images.;
- *Docker containers.* Docker containers are the virtual instances in which the applications are running. Each Docker container contains the running application and all of its dependency files, such as code, system libraries, system tools, and so on.

Users (i.e., Docker clients) can send Docker commands to the Docker daemon, a process running on a host machine to manage and manipulate containers, using the terminal or the remote application programming interface (API). After receiving the commands, the Docker daemon will manipulate the containers and the images on the host machine accordingly, such as by using the images to create containers, pulling images from the remote registry, pushing images to the remote registry, and so on.

III. The Container-based Cluster Management System

Many leading IT companies have integrated Docker into their products, including the Google Cloud Platform, IBM Bluemix, Microsoft Azure, and so on. Although container-based virtualization has brought about advancements in virtualization technology, it is complicated to manipulate a vast number of containers running on a cluster made up of servers. Thus, there is a need for container-based cluster management and orchestration tools. Kubernetes [3] is such an instrument, representing an open source Docker container cluster management tool that was originally developed by Google. So far, a number of IT companies, including IBM, Microsoft, Red Hat, and so on, have supported and contributed to the Kubernetes project. Given its widespread use, Kubernetes seems to be the current trend in the IT industry.

To follow this trend, our work aims at developing a monitoring platform for dynamic resource provisioning on a Kubernetes-based environment. As shown in Figure 1, our system architecture contains one master and multiple nodes, where each node is either a physical or a virtual machine that provides the runtime environment for containers. The node components can be described as follows:

- *Pod.* One or more containers can be grouped into a pod, which is a basic deployment unit that can be operated and managed in a Kubernetes cluster. Each pod can be seen as a virtual server and will be assigned a virtual IP address by the master;
- *Kubelet.* Kubelet is the Kubernetes node agent, which runs on each node of the system. It monitors the pod specifications through the master and maintains these pods on the node. Furthermore, it is also responsible for reporting node events, pod status, and the resource utilization of a node to the master;

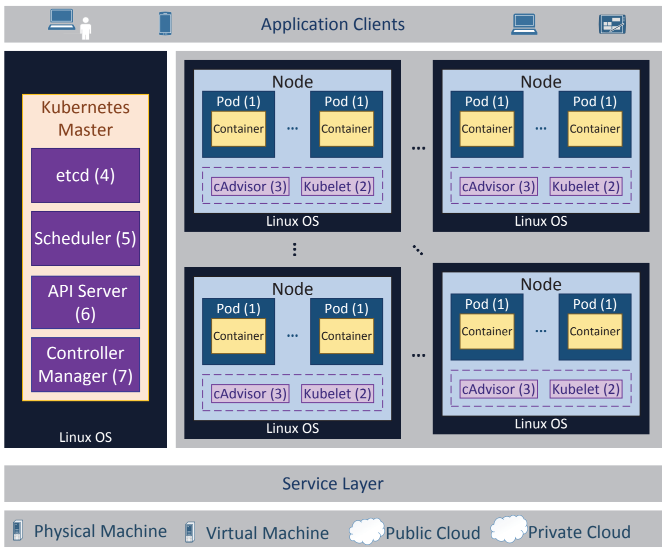


Fig. 1: The system architecture.

- *cAdvisor*. cAdvisor is an open-source container resource usage monitoring tool that has been integrated into Kubelet binary. It automatically gathers the information on resource utilization of all the containers in the node.

The master is the central controller of the system. It contains four Kubernetes master components—etcd, Scheduler, API Server, and Controller Manager—which are used to control and manage the nodes and the containers in the system. Briefly, the functions of the four master components in our system are as follows:

- *etcd*. As a storage component, etcd is used to store the state of the system, allowing the other master components to synchronize themselves to the desired state by watching etcd;
- *Scheduler*. The scheduler is responsible for scheduling each pod on a node in the system;
- *API Server*. The API server is responsible for receiving commands and manipulating the data for Kubernetes objects (such as pods) accordingly in the system. Users can send commands to the API server by using the Kubernetes command line interface (CLI), kubectl;
- *Controller Manager*. The controller manager monitors etcd and regulates the state of the entire system. In other words, if the state of the system changes, the controller manager will force the system into the desired state using the Kubernetes API.

Because our system is based on Kubernetes, some terms and mechanisms of Kubernetes need to be illustrated; these are described as follows:

- *Kubernetes namespaces.* The Kubernetes namespace is used to partition the user-created Kubernetes objects into a named group. Thus, each application can be run into different Kubernetes namespaces to prevent them from affecting each other;
- *Kubernetes services.* A Kubernetes service is used to identify which pods used to execute the same application should be grouped together. When traffic arrives, it will be recognized as its corresponding service IP, and then be redirected to one of its corresponding backet the term "service" represents the "Kubernetes service" below);
- *Replication controllers.* The Kubernetes replication controller is used to control the number of pods that are executed for an application in the system; and
- *Labels and label selectors.* Labels are key and value pairs that are used to group each Kubernetes objects, such as pods, services, and so on, into sets. Label selectors are used to determine the subset of the Kubernetes objects. For example, suppose that a Kubernetes service has two label selectors, key1/value1 and key2/value2; the label set of any pod will then contain these two key and value pairs belonging to this service.

IV. Implementation of the Dynamic Resource-Provisioning Procedure

To realize dynamic resource provisioning, our system implements the Monitor-Analyze-Plan-Execute (MAPE) model [4]. Therefore, our system contains the four modules—Monitor, Data Aggregator, Resource Scheduler and Pod Scaler—which correspond to the four phases of the MAPE loop, respectively. The implementation of these four modules are described in the following subsections.

A. The Monitor Module

The monitor module is used to monitor the metrics of the overall system status. These can be used as the input data of the resource-provisioning algorithm. We divide the monitor module into two parts, namely system resource metrics monitoring and application performance metrics monitoring, which are described below.

1) System resource metrics monitoring: To achieve system resource metrics monitoring, we adopt three existing open source technologies in our system, as follows:

- *Heapster v.0.19.1.* Heapster [5] is a container-based cluster-monitoring tool for Kubernetes. It gathers the resource utilization information of each node and then exports the gathered information into a storage backend.;
- *InfluxDB v.0.9.4.1.* InfluxDB [6] is a database designed to store time series data. In our system, InfluxDB is used to store the resource utilization information collected by Heapster; and
- *Grafana v.2.1.0.* Grafana [7] is an application used for time-series data visualization. It will query the time series data from the database and then display these data in graphs.

The implementation of system resource metrics monitoring is as shown in Figure 2. Specifically, Heapster is the main monitoring component that interacts with InfluxDB and Grafana for database access and visual presentation. Like other applications, Heapster, InfluxDB, and Grafana are all deployed within pods. In our system, they are deployed separately in

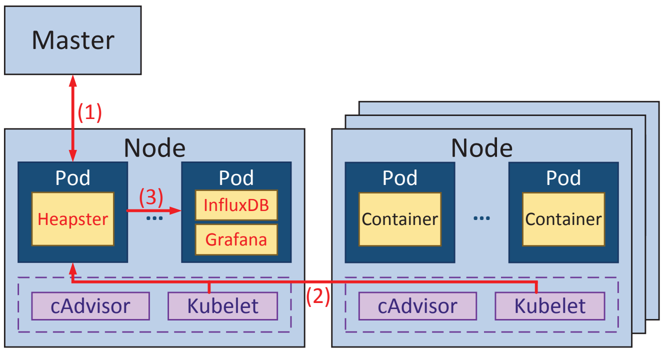


Fig. 2: System resource metrics monitoring.

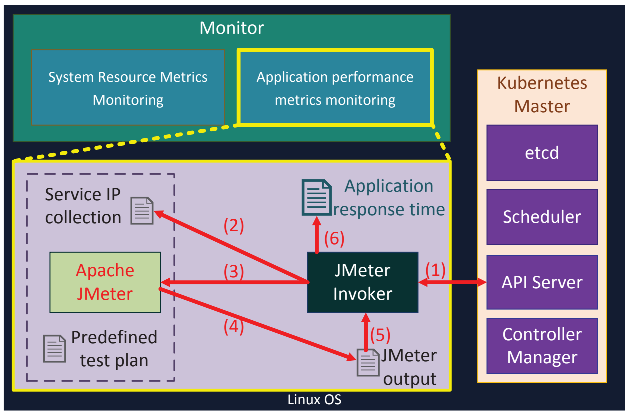


Fig. 3: Application performance metrics monitoring.

three containers, where the Heapster container is placed in one pod, while the other two containers are in another pod. In such a deployment, however, for Heapster to handle the system resource metrics monitoring, we have to let Heapster have the master IP address and the service IP address of the pod which contains the InfluxDB.

After the deployment of Heapster, InfluxDB and Grafana, the system resource metrics monitoring uses the following process:

Step 1. Heapster will ask the master for the list of nodes in the system using the Kubernetes API;

Step 2. In each node, Kubelet gathers the resource utilization information for the whole machine and the containers, which are deployed on that node, by using cAdvisor.

Step 3. After receiving the resource-utilization information concerning each node, Heapster will expose these data into the database, called "k8s" in the InfluxDB. The Grafana deployed in our system will send queries to retrieve the data stored in the "k8s" database and display these data in graphs.

2) Application performance metrics monitoring: Here, we consider the application response time as a representative application performance metric. To monitor application response time, we install Apache JMeter v.2.13 [8] in our system. Apache JMeter is an open source application written in Java. It is designed for workload testing and performance measurement of multiple kinds of servers, such as web servers, ftp servers, database servers, mail servers, and so on. Users can describe the behavior of workload testing in a JMeter test plan and save the test plan as a JMX file. By executing JMeter with the test plan file, JMeter will send the requests to the designated application accordingly. Besides, JMeter can output the information of each requests into a CSV file. Users can employ this output file to analyze the performance of the tested application.

To implement the application performance metrics monitoring, we implement a program, JMeter Invoker, to invoke Apache JMeter and make its behavior in accordance with our expectations. The steps of application performance metrics monitoring are shown in Figure 3; They are as follows:

Step 1. We use the Kubernetes CLI, kubectl, to retrieve the current information about all the services and their corresponding virtual IP addresses and ports throughout the system from the master. The master will return the data in JSON format;

Step 2. JMeter provides some configuration elements that can be used to set up default values or variables. In our JMeter Invoker program, we use CSV Data Set Config to make JMeter send requests to the current service IPs in the system.;

Step 3. We have defined a JMeter test plan file that makes JMeter send 100 requests to each deployed application. We propose that the average interarrival time of these 100 requests in the test plan is three seconds. JMeter will then send requests according to the predefined JMeter test plan to the deployed applications using the service IPs, which are saved in the file generated in step 2;

Step 4. We make JMeter report the results of the requests in a designated file in CSV format. In this file, each line shows the information, including the application URL, response time, response code, and so on, of an individual request;

Step 5. After JMeter finishes sending requests and generates the result file, our JMeter Invoker program will then load the result from the JMeter result file and calculate the average response time for all of the deployed applications;

Step 6. Finally, the JMeter Invoker program outputs the average response time of each deployed application into a file in JSON format.

These six steps will be executed in an infinite loop. Thus, each iteration will update the application response time in the final output file.

B. The Data Aggregator Module

The data aggregator is used to integrate the system status (including nodes and pods), system resource metrics, and application performance metrics. The operation of the data aggregator is shown in Figure 4. The requisite steps are as follows:

Step 1. Fetching information in our system using the

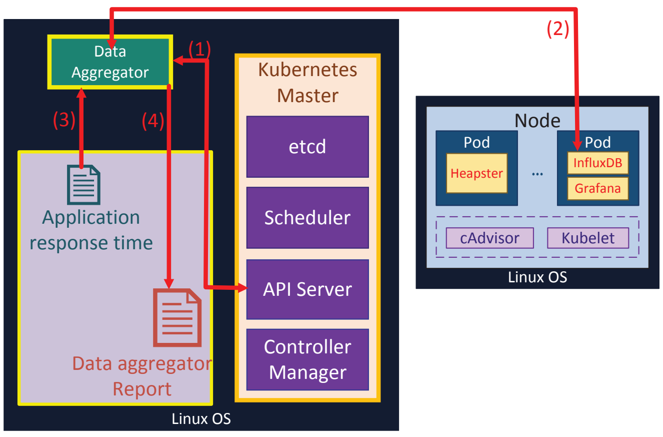


Fig. 4: The operation of data aggregator module.

Kubenetes CLI, kubectl. The information concerning the nodes, namespaces, pods, and services is retrieved from the overall system. For each namespace, we compare the label of each pod and each service and then record the mapping of pods and services;

Step 2. We use the InfluxDB HTTP API to send the InfluxDB Query Language (InfluxQL) to the k8s database to retrieve the stored metrics values. The format of the InfluxDB HTTP API is shown as follows:

```
curl -G 'http://<InfluxDB IP>:8086/query?pretty=
    true' --data-urlencode "db=k8s" --data-
    urlencode "q=<InfluxQL query>"
```

We use the InfluxQL to list the metric values every 10 seconds over the past 6 hours and retrieve the newest value.

Step 3. Fetching the application performance metrics from the output of the JMeter Invoker output file; and

Step 4. Integrating all of the information gathered in the previous steps into a result file in JSON format. The integrated output file is divided into two parts: The first part is the server information in the system, while the other is the application information in the system.

C. The Resource Scheduler Module

The resource scheduler is used to implement the resource-provisioning algorithm. We have implemented a simple resource-provisioning algorithm, algorithm 1, as the default algorithm in our system. The CPU utilization ratio of a pod is its current CPU usage divided by its CPU limit. U_{cpu} represents the arithmetic mean of the CPU utilization ratios of all of the pods that are executed for a distinct application. Similarly, U_{mem} represents the arithmetic mean of the memory-utilization ratios of all the pods that are executed for a distinct application. According to this default algorithm, if the U_{cpu} or U_{mem} of an application in the system is greater than 70%, the application is considered in a heavy-load period. Then, we increase one pod for that application. If the U_{cpu} and U_{mem} of an application in the system are both less than 40%,

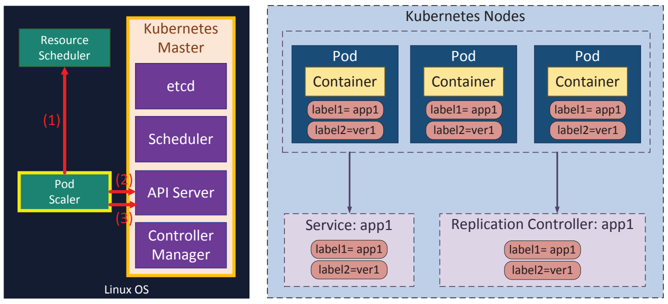


Fig. 5: Pod Scaler.

the application is considered in a light-load period. Therefore, we decrease one pod for the application. Algorithm 1 can be replaced by a newly designed algorithm.

Algorithm 1 Default Resource Provisioning Algorithm

```
Input: The set of applications deployed in our system, $A_n$;
Output: The number of pods of each application;
1: for each application $A_i$ deployed in our system do
2:     $U_{cpu} \leftarrow$ the average CPU utilization ratio of $A_i$
3:     $U_{mem} \leftarrow$ the average memory utilization ratio of $A_i$
4:     $N_{pod} \leftarrow$ the number of pods of $A_i$
5:     if $U_{cpu} \geq 0.7$ OR $U_{mem} \geq 0.7$ then
6:         $N_{pod} = N_{pod} + 1$
7:     else if $U_{cpu} \leq 0.4$ AND $U_{mem} \leq 0.4$ then
8:         if $N_{pod} > 1$ then
9:             $N_{pod} = N_{pod} - 1$
10:        end if
11:    end if
12:    PODSCALER($A_i$, $N_{pod}$)
13: end for
```

D. The Pod Scaler Module

The pod scaler module is responsible for scaling the applications according to the resource provisioning strategy determined by the resource scheduler. Figure 5 shows the operation of the pod scaler. The steps are further described as follows:

Step 1. Receiving the dynamic resource-provisioning strategy from the resource scheduler module;

Step 2. Asking the master for all the replication controller information and then parsing this information to identify the corresponding replication controller. We compare the labels of each pod and each replication controller to find out the corresponding pods and services of a replication controller; and

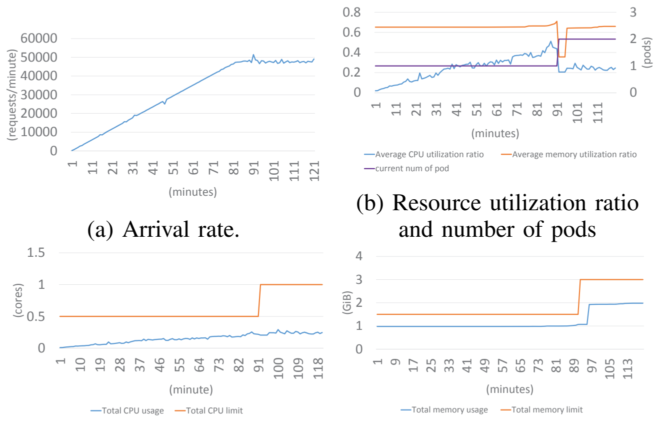
Step 3. Using the kubectl command to set a new size for the replication controller of each application. This will generate the resource provisioning according to the resource-provisioning strategy.

V. Experimental Demonstration

We used OpenShift Origin [9] as our experiment environment. This is an open source cloud application platform

Software	Version
OpenShift Origin	v1.2.0-rc1
Kubernetes	v1.2.0-36-g4a3f9c5
Docker	1.9.1
Operating System	CentOS Linux release 7.1.1503

TABLE I: System environment.



(a) Arrival rate.

(b) Resource utilization ratio and number of pods

(c) CPU.

(d) Memory

Fig. 6: Experiment: the ticket monster application.

proposed by Red Hat that was developed based on Docker (for container packaging) and Kubernetes (for cluster management). Following the OpenShift Origin requirement, we installed CentOS as the OS onto each machine. OpenShift Origin allows us to use the Kubernetes API directly to control all of the Kubernetes components. In our experiment environment, we had one master and one node that were both installed on the same machine. The software versions are provided in Table I. We invoked the dynamic resource-provisioning procedure every 1 minute.

Over this experiment environment, we deployed a JBoss sample application, Ticket Monster [10], as the testing application. Ticket Monster is an online ticketing demo application that users can use to book tickets for concerts, sports games, and so on. We wrapped the application in a Docker image. We then deployed the Ticket Monster into a pod using this image. We used JMeter on another machine to send the HTTP request to the Ticket Monster application. The duration of the experiment was 2 hours; during this time, we made the arrival rate of HTTP requests increase linearly. The Ticket Monster received nearly 50,000 requests per minute at the end of the experiment, as shown in Figure 6a. The comparison of the CPU utilization ratio, memory utilization ratio, and number of executing pods is shown in Figure 6b. Here, we can observe that when the memory utilization reached 70%, the system increased by one pod. Figures 6c and 6d show the relationship of usage quantity and limit quantity of CPU and memory, respectively.

VI. Conclusions

This paper developed a generic platform to facilitate dynamic resource provisioning over Kubernetes, while resolving the existing mechanism's deficiencies. Specifically, our platform has the following features:

- *A comprehensive monitoring mechanism.* Unlike the existing Kubernetes built-in mechanism that only considers CPU utilization, our platform takes both system resource utilization and application QoS metrics into account. Thus, a rather complete set of monitored metrics can be exposed to the resource-provisioning algorithm, for better provisioning decisions;
- *Deployment flexibility.* In our platform, we modularize the four steps of dynamic resource-provisioning, so that the users can easily deploy a newly designed algorithm to replace an existing one without affecting other modules;
- *Automatic operation.* After the users deploy a desired dynamic resource-provisioning algorithm, the resource-provisioning operation will be applied automatically and periodically to all the running applications in our platform following a user-defined time interval.

VII. Acknowledgment

Shun-Ren Yang's work was supported in part by Chunghwa Telecom, and by Ministry of Science and Technology (MOST), R.O.C., under grant numbers MOST 105-2221-E-007-089, MOST 105-2218-E-009-029. Phone Lin's work was supported in part by Chunghwa Telecom, and by Ministry of Science and Technology (MOST), R.O.C., under grant numbers MOST 106-2923-E-002-005-MY3, MOST 104-2923-E-002-005-MY3, MOST 103-2221-E-002-249-MY3, MOST 103-2221-E-002-152-MY3, MOST 105-2627-E-002-001, MOST 105-2622-8-009-008, MOST 105-2218-E-002-013, MOST 105-2218-E-009-029, MOST 105-2511-S-002-006-MY2, MOST 106-2221-E-002-031, MOST 106-2218-E-002-029.

References

[1] Cloud Foundry Foundation. Hope Versus Reality: Containers In 2016. Technical report, June 2016. URL: https://www.cloudfoundry.org/learn/2016-container-report/.
[2] Docker. [Online]. Available: https://www.docker.com/.
[3] Kubernetes. [Online]. Available: http://kubernetes.io/.
[4] E. Casalicchio and L. Silvestri. Architectures for autonomic service management in cloud-based systems. *Computers and Communications (ISCC), 2011 IEEE Symposium on*, pages 161–166, June 2011.
[5] Heapster. [Online]. Available: https://github.com/kubernetes/heapster.
[6] Influxdb. [Online]. Available: https://influxdata.com/time-series-platform/influxdb/.
[7] Grafana. [Online]. Available: http://grafana.org/.
[8] Apache jmeter. [Online]. Available: http://jmeter.apache.org/.
[9] Openshift origin. [Online]. Available: https://www.openshift.org.
[10] Ticket monster tutorial. [Online]. Available: http://developers.redhat.com/ticket-monster/.