# Auto-Scaling in Kubernetes-based Fog Computing Platform

Wei-Sheng Zheng and Li-Hsing Yen

Department of Computer Science, National Chiao Tung University, Taiwan
`kweisamx0322@gmail.com`, `lhyen@nctu.edu.tw`

**Abstract.** Cloud computing benefits emerging Inter of Things (IoT) applications by providing virtualized computing platform in the cloud. However, increasing demands of low-latency services motivates the placement of computing platform on the edge of network, a new computing paradigm named fog computing. This study assumes container as virtualized computing platform and uses Kubernetes to manage and control geographically distributed containers. We consider the design and implementation of an auto-scaling scheme in this environment, which dynamically adjusts the number of application instances to strike a balance between resource usage and application performance. The key components of the implementation include a scheme to monitor load status of physical hosts, an algorithm that determines the appropriate number of application instances, and an interface to Kubernetes to perform the adjustment. Experiments have been conducted to investigate the performance of the proposed scheme. The results confirm the effectiveness of the proposed scheme in reducing application response time.

**Keywords:** Container · Fog Computing · Kubernetes · Scalability.

## 1 Introduction

Internet of thing (IoT) technology supports the connectivity of smart devices to the Internet. A typical IoT application architecture is a fleet of wireless sensors or autonomous vehicles connected to a central cloud in the Internet, where an IoT application server running to collect data from or send instructions to these devices. This architecture suffers from excessive latency between the server and devices and also imposes high traffic load on the backhaul network. Therefore, when latency is a key parameter to the IoT application or when numerous IoT devices are involved, it is needed to place IoT servers in the vicinity of IoT devices. This calls for fog computing.

Fog computing deploys cloud service on the edge of the Internet, i.e., to the proximity of cloud users. For Infrastructure as a Service (IaaS), cloud service is embodied by virtual machines (VMs) or containers. Container is a light-weight virtualization technology that uses cgroups and Linux namespaces to isolate execution environment of applications. Compared with VMs, containers consumes less resource, has a lower loading/starting time, and is easier to manage and control. For this reason, there have been some approaches using container technology to build IoT platform [3, 5].

Docker is a popular container management software that manages containers hosted by a single machine. Ismail et al. [8] used Docker to deploy an edge computing platform. Bellavista and Zanni [2] proposed fog-oriented framework with Docker container for IoT applications. Their experiments demonstrate the feasibility and scalability of fog-based IoT platform.

However, Docker is not suitable for managing a cluster of containers spanning a bunch of machines. Kubernetes [9] (*k8s* for short) manages a cluster of containers that span multiple physical hosts. It cooperates with container management software such as Docker and Rocket [11] to control and manage physically scattered containers. Tsai et al. [12] built a fog platform with Raspberry Pi and Kubernetes, on which TensorFlow was deployed and tested.

*Auto-scaling* is a mechanism that monitors the load status of all application instances and accordingly adjusts the number of instances. Without this, some applications may not fully utilize precious resource allocated to them while other heavily-loaded applications may need resource more than allocated to alleviate and distribute the load. Kubernetes has a native function for container auto-scaling. But it considers only CPU utilization and thus only applies to computation-intensive applications. In fact, whether an application is lightly or heavily loaded should be specific to the application.

Our work took Kubernetes as an orchestrator that instantiates, manages, and terminates containers in multiple-host environments for fog applications, where each host acts as a fog node. On this platform, we developed a dynamic auto-scaling scheme that strikes a balance between resource consumption and application load. Our scheme collects load status from fog nodes, computes an appropriate number of application instances with respect to the load, and cooperates with Kubernetes to adjust the number of application instances. We conducted experiments to compare the performance of the proposed design with with the native scheme. The results indicate that the proposed solution significantly reduces application response time.

The rest of this paper is organized as follows: Section 2 reviews related literature and background. Section 3 details the proposed approach for the fog computing. Section 4 presents our experimental results and Section 5 concludes this work.

## 2 Background and Related Work

Although fog computing differs from cloud computing in many ways, fog computing is not to replace cloud computing because fog platform may not have enough computation resource for some applications. Therefore, we may integrate fog computing and cloud computing to create a more comprehensive solution [10].

Yu et al. [13] proposed a framework for fog-enabled data processing in IoT systems to support low-latency service requested by real-time data applications. Their framework preprocesses sensory data uploaded from sensors at the fog computing platform before further forwarding them to the cloud. This approach effectively reduce the amount of data to be forwarded.

Hu et al. [7] considered running face identification and resolution scheme, which is computation-intensive, on fog platform. The fog-based resolution scheme efficiently performs the designated task. Hao et al. [6] identified some research challenges and problems with fog computing.

As a high-scalability system for OpenStack, ref [1] used a master node that controls the number of VMs running a certain application. The master node also severs as a single entry point for all requests for the application. All requests coming to the master node are queued and then dispatched by the master node to a VM. They also designed automatic scaling-up and scaling-down algorithm, which makes the platform stable and load balance. The problem of this approach is that the master node then becomes a performance bottleneck. We needs a mechanism that separates auto-scaling controller from traffic entry points.

Chang et al. [4] proposed an approach to monitoring Kubernetes container platform. The approach includes a monitoring mechanism that provides detailed information like utilization ratio of system resource and QoS metrics of application. The information enables an sophisticated resource provisioning strategy that performs dynamic resource dispatch. However, the system model used only one machine and did not explain the setting of their parameters.

## 3 A New Dynamic Fog Computing Architecture

We propose a fog computing platform based on a collection of physically distributed containers that is orchestrated by Kubernetes. In this platform, an application instance runs in a single container. More than one application instances may be instantiated on several physical hosts. This is to distribute the load of the application and also extend the service coverage of the application. However, as IoT devices may roam or dynamically participate in and out, the distribution of application instances may not match the distribution of requests coming from devices. As a result, loads on application instances may not be even. The main purpose of our design is to dynamically probe the load status of each application instance, based on which the most appropriate number of application instances that matches the current load and request status can be calculated. With this information, our design then communicates with Kubernetes for the adjustment of application instances, hopefully resulting in a better instance distribution that matches the current request needs from devices.

### 3.1 System Architecture

*Pod* and *Service* are two features of Kubernetes that are essential to the comprehension of our work. Pod is a package of one or containers that share common network namespace, storage, and some policies for restart and healthy check. Pod is the basic control and management unit of Kubernetes and each Pod has its own life cycle. In our work, we always run a single application instance in one Pod. Therefore, $n$ Pods should be created if we should run $n$ application instances.

Service helps dispatch requests to target Pods. Multiple Pods may be identified by outside users using a single IP address. When a packet destined to some IP address is received by the system, any Pod that is configured with the destination IP address can serve the request with equal probability. Kubernetes dispatch packets to all candidate Pods in a round-robin manner, which helps system evenly distribute requests to all serving Pods.

Kubernetes is a client-server architecture that consists of *Master* and *Nodes*. Master controls the whole kubernetes cluster, stores information and opens API for other users or nodes. Nodes is under the control and management of Master.

Master has four components: *Etcd*, *Controller Manager Server*, *Scheduler*, and *APIServer*. Etcd is a distributed key-value store that keeps data across all machines in a reliable way. Controller Manager Server handles any situation of the cluster to ensure stability of Kubernetes. Scheduler decides by which node a newly created Pod is to be hosted. It keeps track of node status, like CPU and memory usage, to select a suitable node for hosting the Pod. Finally, APIServer exports all function using RESTful API for information gathering and operation request.

Kubernetes Node consists of two components: *Kubelet* and *Kube-proxy.* Kubelet running on every node collects node information, including data, volume, image, status for containers, then connects to APIServer for information synchronization with Master. Kubelet also executes instructions from Master. Kube-proxy handles network routing for the accomplishment of service. It also supports basic load balance (round-robin request dispatch).

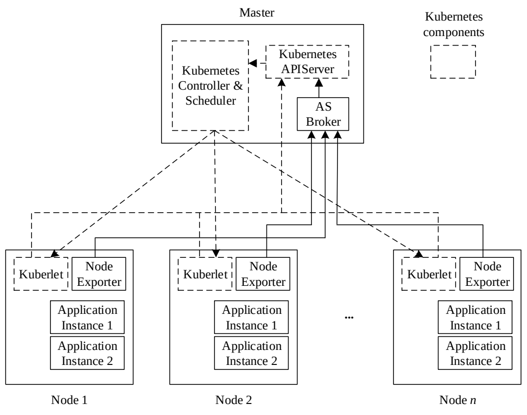

**Fig. 1.** System Architecture

Kubernetes has a native mechanism for auto-scaling (needs installing heapster) that considers only CPU usage. Users could specify the maximal number of application instances, but the actual number of application instances activated is under the control of Kubernetes. We addressed this issue by developing *Auto Scaling Broker* (AS Broker) to dynamically adjust the number of Pods by users.

Fig. 1 shows the whole architecture in including Kubernetes and AS Broker.

## 3.2 Design of AS Broker

We did not modify the round-robin packet dispatch policy of Kube-proxy, though this policy does not consider the current loads of nodes. We also left Scheduler untouched, so Pod placement is also handled by Kubernetes. However, we developed AS Broker to bypass load status reported by Kubelet to Master as we aim at a customized auto-scaling scheme.

AS Broker is a service running in a Pod on Master. It communicates through APIServer with Controller and Scheduler of Kubernetes to obtain a list of nodes where application instances are currently running. It then collects node information from all nodes. It the number of application instances should be adjusted, it then sends a request through APIServer for Pod number adjustment.

There are three major tasks of AS Broker: getting machine information from nodes, estimating an appropriate number of application instances, and requesting Master to exercise the result. For the first task, we run an open-source software *Node Exporter* in a Pod on each node to get information like CPU and memory usage from the physical machine on which it is running. To enable outside access of the information locally kept by Node Exporter, we opens a port on each Pod running Node Exporter so that AS Broker can send a HTTP GET to get node information there. The information consists of raw data of the node status, based on which we can get CPU usage ratio as defined in (1).

$$CPU_{usage} = (usertime + systemtime + \ nicetime)/\mathit{\Delta}, \tag{1}$$

where *usertime* and *systemtime* are the CPU time spent on user and kernel modes, respectively, *nicetime* is the CPU time spent on adjusting the program scheduler or setting priority for programs, and $\mathit{\Delta}$ is the time interval of probe.

In a large-scale system, deploying exactly one Pod running Node Exporter on each node manually may not be viable. We use function *DaemonSet* of Kubernetes to automate this task.

AS Broker probes node information every $\mathit{\Delta}$ seconds, and runs Algorithm 1 every $T_s$ seconds to determine the best number of application instances $n$. If $n$ is different from the current number, AS Broker then asks Kubernetes Controller through APIServer to keep $n$ instances.

The following notations are for Algorithm 1. Let $S$ be the set of all nodes that host the target application instances. For every node $i \in S$, let $u_i^{\mathrm{c}}$ be the CUP utilization ratio and $u_i^{\mathrm{m}}$ be the memory usage. Define $T_{\mathrm{c}}$ and $T_{\mathrm{m}}$ be the thresholds of CPU utilization and memory usage to trigger an adjustment. Parameter $\alpha$ is a factor that indicates the weighting between CUP utilization and memory usage.

It should be high (close to 1) for computation-intensive applications. Parameter $\beta$ is an adjustment factor that keeps the value of $n$ not exceeding the maximal number of Pods allowed for the applications.

**Algorithm 1** best-instance-number($S$, $\{u_i^{\mathrm{c}}\}$, $\{u_i^{\mathrm{m}}\}$)

```
    Parameter
$\alpha$: weighting factor between $U_{\mathrm{c}}$ and $U_{\mathrm{m}}$
$\beta$: adjustment factor
    Output
$n$: number of Pods (application instances)

1: if $\exists i \in S, u_i^{\mathrm{c}} > T_{\mathrm{c}}$ or $u_i^{\mathrm{m}} > T_{\mathrm{m}}$ then
2:     $U_{\mathrm{c}} = \sum_{i \in S} u_i^{\mathrm{c}} / |S|$            ▷ Avg. CPU utilization
3:     $U_{\mathrm{m}} = \sum_{i \in S} u_i^{\mathrm{m}} / |S|$            ▷ Avg. memory usage
4:     $n = (\alpha * U_{\mathrm{c}} + (1 - \alpha) * U_{\mathrm{m}}) / \beta$
5: else
6:     $n = 1$
7: end if
8:
```

## 4 Experimental Results

Our fog platform consisted of four PCs. Each PC has a 3.2 GHz i5-6500 CPU and 8 GB RAM. The operating system is Ubuntu 16.04LTS. Kubernetes version is 1.6.1. Among them, three PCs were Nodes and the other was Master. AS Broker ran on the Master.

We used another PC to generate and send requests to the fog platform. We used *stress* program to generate CPU and memory load to emulate the processing of requests. Every request took 15-second execution time and 50-MB memory space. The inter-arrival time of requests was an exponential distribution with mean $1/\lambda$, rendering request arrivals a Poisson process with mean arrival rate $\lambda$. All requests were sent toward to an application and directed by Kubelet to a Node where an application instance was running. We varied the value of scaling interval $T_s$ to investigate the impact of $T_s$ on performance. Because instantiating a container took three seconds, the value of $T_s$ was ranged from 10 to 60 seconds. Fig. 2 shows the experiment environment.

We varied the value of $\lambda$ and ran a 600-second trial with $T_s = 20$ for each setting of $\lambda$. Fig. 3 shows cumulative probabilities of the number of Pods for different $\lambda$. We can see that with $\lambda = 1$, 47% of the time there were only six or fewer application instances (Pods). When $\lambda$ was set to 2, the percentage dropped to 30%. It was 10% with $\lambda = 4$. When $\lambda$ was 5, there were 8 or 9 application instances running 80% of the time.

We also tested application response time with and without AS Broker. Fig. 4 shows the result with $\alpha = 0.8$, $T_s = 30$, and $\lambda = 1$. Clearly, though response

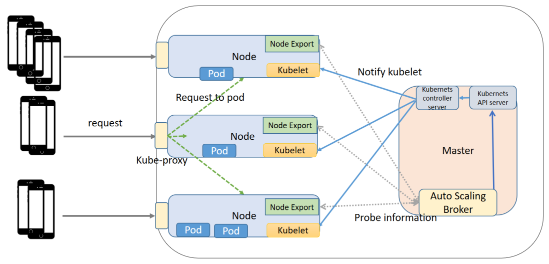

**Fig. 2.** Experiment Environment

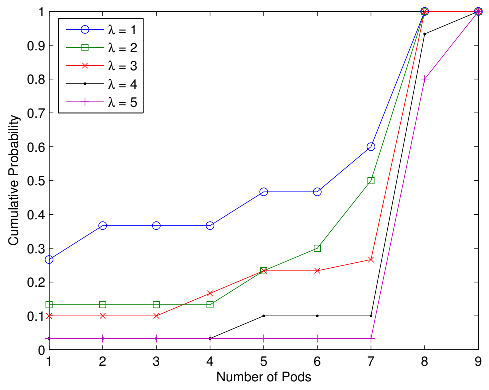

**Fig. 3.** Number of Pods (application instances) with different $\lambda$

time dynamically changes, the result with AS Broker is better than that without almost at every time point. This result demonstrates the effectiveness of the proposed scheme.

## 5 Conclusions

We has proposed an auto-scaling scheme for Kubernetes-based fog computing platform. It uses an open-source software Node Exporter running in a Pod on each node to get machine information. AS Broker collects machine information to determine the most appropriate number of application instances. It then asks Kubernetes to keep the desired number of Pods. Experimental results confirm the effectiveness of the proposed scheme in reducing application response time.

## References

1. de la Bastida, D., Lin, F.J.: OpenStack-based highly scalable IoT/M2M platforms. In: IEEE Int'l Conf. on Internet of Things. Exeter, UK (Jun 2017)

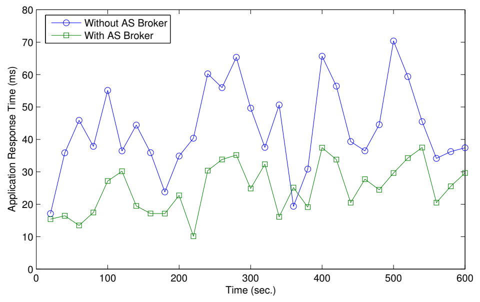

**Fig. 4.** Application response time with and without AS Broker ($\alpha = 0.8$, $T_s = 30$, $\lambda = 1$)

2. Bellavista, P., Zanni, A.: Feasibility of fog computing deployment based on Docker containerization over RaspberryPi. In: Proc. 18th Int'l Conf. on Distributed Computing and Networking (Jan 2017)
3. Brogi, A., Mencagli, G., Davide Neri, J.S., Torquati, M.: Container-based support for autonomic data stream processing through the Fog. In: Euro-Par 2017: Parallel Processing Workshops. pp. 17–28 (2017)
4. Chang, C.C., Yang, S.R., Yeh, E.H., Lin, P., Jeng, J.Y.: A kubernetes-based monitoring platform for dynamic cloud resource provisioning. In: IEEE Global Communications Conference (Dec 2017)
5. Dupont, C., Giaffreda, R., Capra, L.: Edge computing in IoT context: Horizontal and vertical linux container migration. In: Global Internet of Things Summit (Jun 2017)
6. Hao, Z., Novak, E., Yi, S.: Challenges and software architecture for fog computing. IEEE Internet Computing **21**(2), 44–53 (Mar 2017)
7. Hu, P., Ning, H., Qiu, T., Zhang, Y., Luo, X.: Fog computing based face identification and resolution scheme in internet of things. IEEE Trans. on Industrial Informatics **13**(4), 1910–1920 (Aug 2017)
8. Ismail, B.I., Goortani, E.M., Karim, M.B.A.: Evaluation of Docker as Edge computing platform. In: 2015 IEEE Conference on Open Systems. Bandar Melaka, Malaysia (Aug 2015)
9. kubernetes: Production-grade container orchestration, https://kubernetes.io/
10. Mebrek, A., Merghem-Boulahia, L., Esseghir, M.: Efficient green solution for a balanced energy consumption and delay in the IoT-Fog-Cloud computing. In: 16th Int'l Symp. on Network Computing and Applications. Cambridge, MA, USA (Oct 2017)
11. Rocket: A security-minded, standards-based container engine, https://coreos.com/rkt
12. Tsai, P.H., Hong, H.J., Cheng, A.C.: Distributed analytics in fog computing platforms using TensorFlow and Kubernetes. In: 19th Asia-Pacific Network Operations and Management Symposium. Seoul, South Korea (Sep 2017)
13. Yu, T., Wang, X., Shami, A.: A novel fog computing enabled temporal data reduction scheme in IoT systems. In: IEEE Global Communications Conference (Dec 2017)