IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Date of publication xxxx 00, 0000, date of current version xxxx 00, 0000.

Digital Object Identifier 10.1109/ACCESS.2017.DOI

ElasticFog: Elastic Resource Provisioning in Container-based Fog Computing

NGUYEN DINH NGUYEN[1], LINH-AN PHAN[1], DAE-HEON PARK[2], SEHAN KIM[2], AND TAEHONG KIM[1]

[1]School of Information and Communication Engineering, Chungbuk National University, Cheongju, Republic of Korea
[2]Electronics and Telecommunications Research Institute, Daejeon, Republic of Korea

Corresponding author: Taehong Kim (e-mail: taehongkim@cbnu.ac.kr).

This work was partially supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. NRF-2019R1F1A1059408) and Institute of Information & communications Technology Planning & Evaluation (IITP) grant funded by the Korea government (MSIT) (No.2018-0-00387, Development of ICT based Intelligent Smart Welfare Housing System for the Prevention and Control of Livestock Disease).

ABSTRACT The recent increase in the number of Internet of Things (IoT) devices has led to the generation of a large amount of data. These data are generally processed by cloud servers because of their high scalability and ability to provide resources on demand. However, processing large amounts of data in the cloud is an impractical solution for the strict requirements of IoT services, such as low latency and high bandwidth. Fog computing, which brings computational resources closer to the IoT devices, has emerged as a suitable solution to mitigate these problems. Resource provisioning and application orchestration are two of the key challenges when running IoT applications in a Fog computing environment. In this paper, we present ElasticFog, which runs on top of the Kubernetes platform and enables real-time elastic resource provisioning for containerized applications in Fog computing. Specifically, ElasticFog collects network traffic information in real time and allocates computational resources proportionally to the distribution of network traffic. The experimental results prove that ElasticFog achieves a significant improvement in terms of throughput and network latency compared with the default mechanism in Kubernetes.

INDEX TERMS container, Fog computing, Internet of Things (IoT), Kubernetes, resource provisioning.

I. INTRODUCTION

The concept of the Internet of things (IoT), which connects smart devices to each other through the Internet, has become popular over the past few years. According to [1], the number of smart devices will increase from 20 billion in 2019 to approximately 30 billion in 2023. Because of several powerful features of Cloud computing, such as scalability, on-demand resource allocation, and a pay-as-you-go model [2], the massive data generated from millions of distributed IoT devices are transmitted to centralized servers through the Internet for processing. This design consumes considerable time, bandwidth, and money [3]. With the rapidly growing number of connected devices, this traffic pattern creates pressure on the network and leads to a poor experience for end users, especially for time-sensitive IoT applications, such as augmented reality applications, intelligent transportation systems, and smart cities [4], [5]. Therefore, Cloud computing faces several challenges in meeting the requirements of IoT devices, including stringent latency requirements and network bandwidth constraints [6]. These challenges hinder the prospective benefits of IoT for everyday life.

Fog computing [7], [8] has emerged as a suitable solution to overcome the limitations of the traditional cloud by exploiting computation, network, and storage resources that are close to the end devices. Having the computational resources close to the data generated by IoT devices has several benefits, such as accelerated analysis, lower latency, and better user experience. Although there have been several efforts and discussions toward the development of Fog computing, it is still in its infancy. Existing solutions focus on the basic idea and theoretical analysis, and most of them lack practical solutions for efficient management of IoT applications in Fog computing.

Recently, the use of container [9], which is a lightweight method for creating virtual environments, has been exploited

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3029583, IEEE Access

in both Cloud computing and Fog computing infrastructure. This method is more suitable than the use of virtual machines for Fog computing because it can be easily deployed and has high performance [10]. Containers do not comprise an entire operating system; only the relevant application and its dependencies are bundled into a single package. This feature enables containers to be more efficient and lightweight, and allows for faster deployment [10]. Moreover, container-based microservices for IoT applications are considered as one of the best solutions for minimizing the resource limitation problem in Fog computing [11]. Owing to these advantages, containers have been widely adopted for IoT applications in the Fog computing environment.

However, having numerous IoT applications in Fog computing requires an orchestration tool to deploy and manage containers and their resources effectively. Kubernetes (K8S) [12] is a well-known open-source orchestration platform for container-based applications. It provides various powerful and flexible functions for container orchestration, such as deployment, self-healing, resource management, load balancing, and auto-scaling.

In Fog computing, because of the heterogeneous and limited resource capabilities of the Fog nodes, several Fog nodes communicate and share client requests to handle them effectively [13]. This cluster of Fog nodes provides higher resource capabilities to deploy and execute applications, more convenience in management, and more cost savings. In this study, we focus on solving two problems in the Fog computing environment. First, Fog nodes are geographically distributed in several locations, which causes a delay in communication among the Fog nodes in the cluster. Hence, requests should be handled at the local location as much as possible to minimize network delay. Second, the network traffic accessing the application at each location often fluctuates substantially according to demands on the application. This implies that the resource provisioning for the application on each Fog node requires frequent updates to adapt to the changes in demands on the application. In other words, if a large amount of network traffic accesses the application from a certain location, the corresponding resources at that location should be increased to handle the requests quickly and effectively. Despite its several powerful policies and features, Kubernetes only considers the software and hardware status on each node for managing application resources, which is insufficient to ensure the quality of service in a Fog computing environment. Therefore, we propose an elastic resource provisioning mechanism that exploits the scheduling in Kubernetes and information regarding network traffic in real time. The proposed mechanism aims to allocate resources to each Fog node dynamically in proportion to the amount of network traffic accessing the application at each location. Thus, the mechanism is expected to minimize network latency as well as avoid resource wastage in locations with low workload demand. To enable this feature, ElasticFog is deployed on top of the Kubernetes platform to collect real-time information regarding network traffic at each Fog node and attempt to allocate resources effectively. Through experimental evaluations, we verified the effectiveness of resource provisioning based on real-time network traffic in a Fog computing environment.

The remainder of this paper is structured as follows. Section II presents the related work, while Section III describes the fundamentals of Kubernetes. In Section IV, we describe the proposed elastic resource provisioning method in container-based Fog computing. The performance evaluations are then presented in Section V. Finally, the conclusions of the study are presented in Section VI.

II. RELATED WORK

Recently, there have been several studies regarding resource provisioning and orchestration for IoT applications in the Fog computing environment. For example, Skarlat *et al.* [14] presented a model for resource provisioning in Fog computing. The authors formalized an optimization problem that analyzes resource usage to create a resource provisioning plan and schedules task requests for the services running in Fog nodes. The aim of the study is to minimize network delay using the computational resources in the Fog nodes as much as possible. In [15], task scheduling, task image placement, and other issues are jointly considered to minimize the task completion time in the Fog computing environment. The authors focused on investigating the placement of task images on the storage servers and balancing the workload between end devices and computation servers. Aazam *et al.* [16] presented a resource management method for the services in Fog computing. It estimates the required resources and predicts the resource allocation for services based on the types of services and the user's historical behavior. The goal is to minimize resource wastage in Fog nodes and increase profits for service providers. Paper [17] proposed a dynamic resource allocation strategy for Fog computing based on priced timed Petri nets. The resources in Fog nodes can be classified into several groups. Users can select the satisfying resources autonomously from the provided resources according to the price and time cost of the task. Paper [18] presented methods for efficient resource management and workload allocation in Fog-Cloud computing. These methods are based on learning classifier systems and consider four dimensions: workload, delay, power consumption, and battery status. The target is to optimize the allocation of workload between Fog and Cloud to balance the power consumption and delays at the edge.

An orchestration framework for containerized applications in Fog computing infrastructures is presented in [19]. The framework is developed based on the OpenIoTFog toolkit and Docker Swarm Services. The proposed method can resolve orchestration problems in the Fog environment, such as cluster joining and leaving, application scheduling, and device mapping from Fog nodes to containerized applications. A platform for workload orchestration and resource negotiation in the Fog computing environment is presented in [20]. The platform is based on open-source technolo-

gies (Openstack, Kubernetes, and Foggy), and it considers multiple parameters such as traditional requirements (e.g., RAM and CPU) and non-traditional ones (e.g., bandwidth, location), to select a location for application deployment. In [21], a QoS-aware network resource management framework for containers in Fog computing is proposed. The framework is lightweight and can be used to control the bandwidth of containers on Fog nodes. It provides three scheduling policies for containers: proportional share scheduling, minimum bandwidth allocation, and maximum bandwidth limitation. Paper [22] presented a model that improves the scheduler of containerized services - Diego. It abstracts the dilemma between load balance and application performance by formulating it as an optimization problem and using statistical methods to solve it efficiently. To minimize the network contention in Fog computing, dependent applications, which need to communicate with each other, are deployed near each other on a prioritized basis (e.g., in the same zone because the latency between nodes is small). Paper [23] proposed a container-based task scheduling algorithm to ensure that tasks are completed in time. The scheduling can decide to handle the task in either a Fog node or the Cloud, based on the estimated task execution time. If the tasks are executed in a Fog node, the resource manager reallocates resources to them to complete the tasks within a delay constraint.

Fogernetes [24], which is implemented based on Kubernetes, is a platform for the deployment and management of applications in Fog computing. This platform uses a labeling system to describe the application requirements and the capabilities of Fog nodes. It enables the management and deployment of applications by considering the specific application requirements and different capabilities of heterogeneous Fog nodes. A network-aware scheduling approach is presented in [25], [26] for deploying container-based applications in Fog computing. The authors extended the Kubernetes scheduler to consider network infrastructure status for taking resource provisioning decisions. However, this approach only considers these parameters at the time of initialization when the application is deployed; there is no dynamic adjustment during runtime. Paper [27] presented a scheduling approach to place containerized service chains in Fog-Cloud environments efficiently. The scheduling is implemented as an extension to the default scheduler in Kubernetes. When the application is deployed, the best-fit node for placing the pod is selected based on two policies: *latency-aware* and *location-aware*. For *latency-aware*, the pods that belong to the same *service function chain* can be deployed near each other based on Dijkstra's shortest path algorithm. If *location-aware* is selected, the selection of node is based on minimizing the latency with the *target location* of the pod. Paper [28] proposed an orchestration tool (*ge-kube*) for container-based applications in the geo-distribution of the Fog computing environment. *Ge-kube* relies on Kubernetes and provides two logical components: elasticity manager and placement manager. For elasticity manager, a model-based reinforcement learning approach dynamically scales the number of replicas of the application in real time. The placement manager exploits an optimization problem formulation and a network-aware heuristic, which consider the network delays among computing resources. This policy can allocate a pod to the node whose network delay with other pods is below a critical value.

Although the aforementioned studies resolve some challenges in resource provisioning and application orchestration in Fog computing, they have not yet provided a complete solution for the real-time management of application resources based on network traffic information. In this paper, the powerful Kubernetes platform is exploited to deploy and manage the containerized applications in Fog computing. By leveraging powerful and flexible features of Kubernetes and real-time network traffic from clients accessing the Fog nodes, our proposed approach can effectively allocate and reallocate IoT application resources to adapt to the changes in demands on the application in the Fog computing environment.

III. FUNDAMENTALS OF KUBERNETES

To better understand our proposal, this section details basic Kubernetes architecture and its scheduling mechanism.

A. KUBERNETES ARCHITECTURE

Kubernetes is a well-known open-source orchestration platform for container-based applications. A pod is the most fundamental unit in Kubernetes, and each pod contains one or more containers. The containers in a pod are tightly coupled, share storage, and use a unique IP. The Kubernetes architecture is shown in Fig. 1. There are two kinds of nodes in Kubernetes: master and worker nodes. The master node is responsible for managing and controlling the cluster. There is a single master node by default, but several master nodes can be implemented for high availability. The master node contains four main components: *etcd*, *kube-apiserver*, *kube-controller*, and *kube-scheduler* [12]. Here, *etcd* is a distributed key-value store that Kubernetes uses for the persistent storage of cluster data, including configuration data and the state of the cluster. The *kube-apiserver* handles all requests that interact with the control plane of the cluster. This component authenticates the requests and updates the corresponding information in *etcd*. Finally, the *kube-controller* runs control loops that continuously watch the current state of the cluster to ensure that the current state matches the desired state. For example, the *kube-controller* ensures that the number of running replicas of an application always matches the number of replicas defined in the configuration. The *kube-scheduler* watches unscheduled pods and selects the best fit node for them to run on. To find the best fit node, it considers multiple factors, such as resource requirements, available resources, affinity, and anti-affinity rules.

In the Kubernetes cluster, worker nodes are used to run containerized applications, which are deployed in the form of pods. A worker node consists of three main components, namely *container runtime* (e.g., Docker [29]), *kubelet*, and

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3029583, IEEE Access

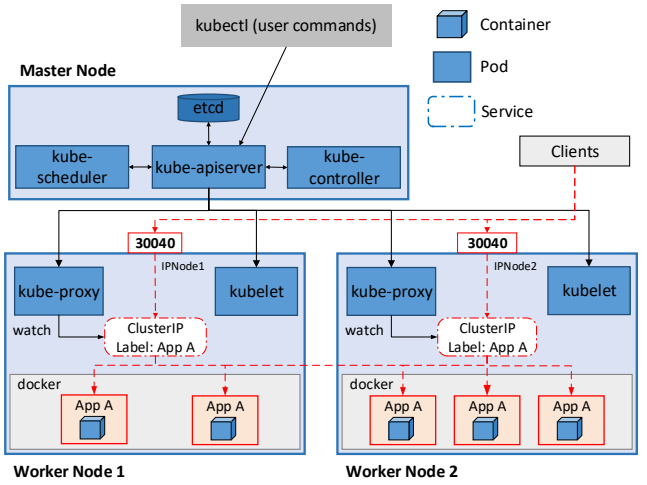

FIGURE 1. Kubernetes architecture.

kube-proxy. *Container runtime* is responsible for pulling the container image from a public or private registry and running the container based on the image. *Kubelet* communicates with the master node to receive commands and report pod and node statuses. This component executes the pods on the node and ensures the health of the pods (e.g., by restarting failed pods). Finally, *kube-proxy* maintains the network rules and handles network communications inside and outside of the cluster.

In Kubernetes, it is difficult to access a pod using the pod's IP address because the IP address always changes after a restart. To expose an application to external cluster access, Kubernetes provides a Service that is an abstract layer for a group of pods in the cluster. The Service is bound to a *ClusterIP*, which is a virtual IP address that never changes. A list of backend pods belonging to the same Service is determined using a *LabelSelector*. The traffic entering *ClusterIP* is routed to a backend pod on this list. The selection of the pod to load-balance the traffic among pods depends on the proxy mode configured in the *kube-proxy*. By default, the *user-space* proxy chooses a pod using a round-robin algorithm, and the *iptable* proxy selects a pod at random. The *IP Virtual Server* (*IPVS*) provides more options (e.g., destination hashing, source hashing, and round-robin) to select the pod. However, *ClusterIP* makes the Service reachable only from inside the cluster. To make the Service reachable from outside the cluster, two types of Service are typically used: *NodePort* and *LoadBalancer*. With regard to *NodePort* Service, a static port is opened on each node. Clients can access the Service from outside the cluster using the IP address of the node and the static port. For example, in Fig. 1, a *NodePort* is created (the static port is 30040) for a set of pods that has the label *App A*. Clients can access the Service using the address *IPNode1*:30040. The traffic is then redirected to the *ClusterIP* of the Service and routed to a backend pod that is selected by the *kube-proxy*. The *LoadBalancer* Service is supported if a cloud provider for the Kubernetes cluster is used. The cloud provider provides a public IP address for the Service to make it reachable from outside the cluster. The selection of the backend pod for load balancing depends on the implementation of the cloud provider.

B. SCHEDULING IN KUBERNETES

In Kubernetes, scheduling refers to selecting the most appropriate node for the pod to run on. *Kube-scheduler* is the default scheduler in Kubernetes, and it watches unscheduled pods and selects an optimal node for them to run on. The scheduling process in Kubernetes is illustrated in Fig. 2. The unscheduled pods are added to a waiting queue. The *kube-scheduler* picks a pod in the queue and selects the best node for the pod from the list of running nodes based on *filtering* and *scoring* steps [12]. The *filtering* step finds a set of nodes, which are called feasible nodes. These nodes meet certain scheduling requirements of the pod based on a list of policies. In the *scoring* step, the feasible nodes are ranked to find the most suitable node for the pod according to different priority functions.

The following are some policies that are supported in the *filtering* step in Kubernetes [12]:

- *PodFitsHostPorts*: checks if any port requested by the pod is free in the node.
- *PodFitsResources*: checks if the free resources (e.g., CPU and memory) of the node meet the requirements of the pod. For example, if the pod requests 512 MiB of RAM, the nodes that have less than 512 MiB of free RAM are removed from the list.
- *PodFitsHost*: checks if a pod requires to be deployed in a specific node using the node name.
- *CheckNodeCondition*: checks whether a node is healthy. For example, whether the network is available or the *kubelet* is ready to deploy the pod.

Other policies in the *filtering* step include *PodMatchNodeSelector*, *NoDiskConflict*, and so on. After *filtering* step, there is often more than one node remaining. The *scoring* step is then performed by a set of priority functions to find the best fit node from the feasible nodes. The Kubernetes scheduler calculates a priority score for every node in the list of remaining nodes. The priority score is given by the priority functions. Each priority function has a different weight, and the final priority score of each node is the sum of all weighted score given by the priority functions. After calculating the priority score for all feasible nodes, the node with the highest score is selected as the best fit node. Kubernetes implements several priority functions in the *scoring* step, including [12]:

- *SelectorSpreadPriority*: spreads the pods belonging to the same service across nodes in the cluster.
- *BalancedResourceAllocation*: prioritizes the nodes having a balanced resource allocation (CPU and memory utilization) after the pod is deployed.
- *NodeAffinityPriority*: prioritizes nodes based on node affinity scheduling preferences. For example, the pod

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3029583, IEEE Access

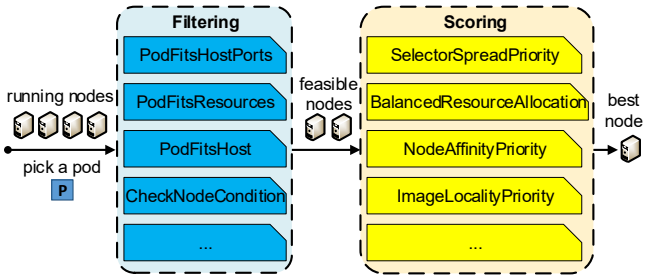

FIGURE 2. Scheduling in Kubernetes.

can be configured for deployment in a specific set of nodes (e.g., in the same location).

- *ImageLocalityPriority*: prioritizes nodes that already contain the requested container images for that pod in local.

Other priority functions provided by Kubernetes include *LeastRequestedPriority*, *ServiceSpreadingPriority*, and so on. The scheduling in Kubernetes is a powerful feature based on two steps that are compliant with a set of different policies. The scheduling decision is based on not only the "hard" requirements (e.g., CPU and RAM) in the pod's configuration but also diverse "soft" requirements (e.g., spreading the pods and balanced resource allocation) to find the best fit node for the pod to run on.

IV. ELASTICFOG

In this section, we discuss the Fog computing architecture based on Kubernetes. Then, we describe the proposed ElasticFog, which dynamically allocates resources for containerized applications in a Fog computing environment in real time.

A. KUBERNETES-BASED FOG COMPUTING ARCHITECTURE

The high-level Fog computing architecture is depicted in Fig. 3. The bottom tier includes several IoT devices that send requests to the upper tiers mainly through wireless gateways that are linked directly with the Fog nodes. The middle tier is the Fog computing layer including Fog nodes that can compute and analyze requests from the IoT devices. The top tier is the Cloud computing layer that includes several servers in a data center [30].

While Fog computing provides several potential advantages, it is still in the early stages of development. Thus, several challenges, such as resource provisioning and application orchestration for millions of IoT services, still require attention. In our architecture, the Kubernetes platform has been leveraged for the application orchestration. The IoT applications are containerized and deployed in the form of pods in the Fog nodes. To achieve high availability and performance, an IoT application can have multiple pod replicas, and Kubernetes can load-balance the traffic among these pods. Because the resource capabilities of the Fog nodes are

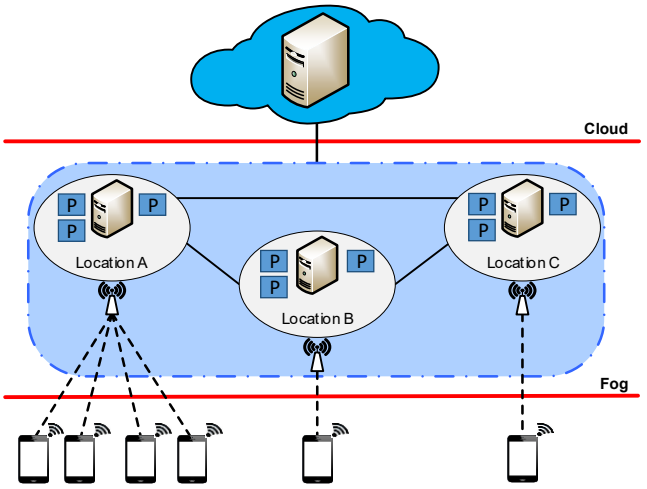

FIGURE 3. Architecture of Kubernetes-based Fog computing.

heterogeneous, limited, and cannot be dynamically scaled, the Fog node may face a lack of required resources to deploy and execute the services. To handle this problem, several Fog nodes in nearby locations (e.g., in the same city, campus, or factory) can communicate and work together as a cluster. In other words, the deployment and execution of an application can be shared among the Fog nodes in a cluster. However, the co-operation of Fog nodes in a Kubernetes cluster also causes latency for redirecting requests to the Fog nodes in other locations. Therefore, it is important to optimize the performance of the cluster by allocating proper application resources at each location to minimize the amount of network traffic that is redirected and handled by Fog nodes in other locations.

Although the Kubernetes scheduler supports several powerful policies, these policies are insufficient for IoT services because of the peculiarities of the Fog computing environment. There are two critical limitations of Kubernetes in the Fog computing environment:

- The pods of an application are distributed among Fog nodes. However, the scheduling mechanism in Kubernetes does not consider the network traffic status to schedule the pods of the application at each Fog node. For example, in Fig. 3, although a large proportion of the network traffic is accessing location A, the pods are still evenly distributed across all locations. Thus, location A may lack the resources to handle the requests immediately and this may cause a significant decrease in the quality of service for clients on that location.
- In the Fog computing environment, the network traffic accessing the application at each location fluctuates substantially according to the density of IoT devices, and the popularity of applications at different times. Therefore, despite the frequent changes in resource demands of the application at each location, Kubernetes lacks the ability to monitor these changes according to

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3029583, IEEE Access


```
apiVersion: extensions/v1beta1
kind: Deployment
metadata:
 name: IoT-app-1
spec:
 replicas: 9
 template:
  metadata:
   labels:
    app: IoT-app-1
  spec:
   affinity:
    nodeAffinity:
     preferredDuringSchedulingIgnoredDuringExecution:
     - preference:
        matchExpressions:
        - key: location
          operator: In
          values:
          - locationA
       weight: 80
     - preference:
        matchExpressions:
        - key: location
          operator: In
          values:
          - locationB
       weight: 20
  containers:
  ...
```

FIGURE 4. Configuration for the preferred rule in nodeAffinity.

the real-time network traffic status.

Considering these two problems, we develop a framework to provide application resources effectively in a Fog computing environment. First, the network traffic status at each location is considered to determine the appropriate resource allocation at each location. For example, in Fig. 3, the number of pod replicas at location A can be increased to handle the large proportion of incoming network traffic. Second, our framework can collect network traffic information frequently in real time and allocate resources dynamically based on this information, to adapt to the changes in network traffic status over time.

B. ELASTIC RESOURCE PROVISIONING IN CONTAINER-BASED FOG COMPUTING

In this section, we describe the proposed elastic resource provisioning method for applications in container-based Fog computing (ElasticFog), using Kubernetes' scheduling features and network traffic information in real time. The goal is to distribute a proper number of pods of the IoT application at each location automatically, based on the distribution of incoming network traffic at each the location in real time.

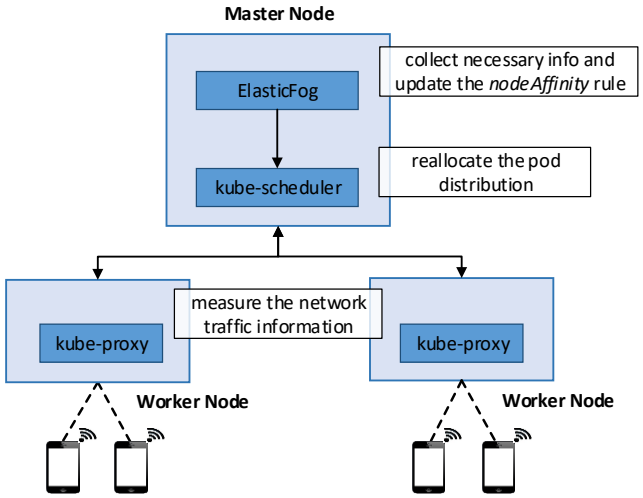

FIGURE 5. Main workflow in the ElasticFog-based framework.

To exploit the information about the incoming network traffic of the application, we define and create a new corresponding Endpoint object to store this information. We use the *user-space* proxy to measure the average of network traffic in bytes per second (Bps) accessing the application at each worker node (Fog node) at predetermined intervals. This value is then stored in the Endpoint object of the application. Finally, the master node can retrieve the network traffic status by reading this Endpoint object.

In Kubernetes, the scheduler provides a priority rule, named *nodeAffinity* (the *NodeAffinityPriority* priority function), which can constrain the place of a pod on specific nodes. There are two kinds of *nodeAffinity* rules: *required* and *preferred*. The *required* rule means that the selected node must be met for the pod's requirements, while the *preferred* rule means that Kubernetes scheduling attempts to enforce the rule. Thus, the nodes that satisfy the *preferred* rule are favored for placing the pod. For the *preferred* rule, a *weight* field is specified for each node, from 1 to 100. The higher the *weight* value, the more preferred the node is. In the Kubernetes cluster, each node can have a label in a key-value form. As shown in Fig. 4, the preferred nodes in the configuration of the *nodeAffinity* rule are determined based on this key-value pair. ElasticFog exploits the *preferred* rule to allocate pods of the application across locations. By using the *preferred* rule, we can ensure high reliability and availability of the application by combining several other beneficial policies provided by the Kubernetes scheduler (e.g., checking the health of nodes, spreading the pods belonging to the same service and balanced resource allocation). To determine the node and its location in ElasticFog, the label of the Fog node is set based on the location of the node. Specifically, the key is *location*, and the value is the location's name. For example, the label of the node belonging to location A is set as *location*: *locationA*. The value of the *weight* field for each location is calculated based on the proportion of

Algorithm 1 Algorithm in ElasticFog

```
/* applicationList: List of applications running in the cluster.*/
/* nodeList: List of nodes in the cluster. */
/* locationInfo: Information about the list of nodes at each location. */
/* appTrafficInfo: Information about network traffic of the applications on the node. */
/* weightInfo: The proportion of network traffic at the location of the application. */
function CALCULATEWEIGHT(application)
    totalTraf = 0
    // calculate the total of traffic on the cluster
    for node in range nodeList do
        totalTraf += appTrafficInfo(node)
    end for
    for location, nodes in range locationInfo do
        locationTraf = 0
        // calculate the total of traffic at the location
        for node in range nodes do
            locationTraf += appTrafficInfo(node)
        end for
        // calculate proportion of traffic of the location
        proportion = (locationTraf / totalTraf)*100
        weightInfo(location) = math.Round(proportion)
    end for
    return weightInfo
end function
function MAIN
    // traverse all the applications running in the cluster
    for application in range applicationList do
        // calculate the weight info for each location
        weightInfo = calculateWeight(application)
        // update the weight info for the locations
        updateNodeAffinity(weightInfo)
    end for
end function
```

incoming network traffic of the application at that location. To accommodate 80% of the incoming network traffic of application "IoT-app-1" at location A and 20% of incoming network traffic at location B, the *weight* value is set to 80 for location A and 20 for location B.

Fig. 5 describes the main workflow in the ElasticFog-based framework. *Kube-proxy* measures the information about the incoming network traffic of the application on each Fog node. ElasticFog can be implemented as a control component in Kubernetes, and it periodically collects up-to-date information about the applications, Fog nodes, and network traffic status. Using this information, ElasticFog calculates the proportion of incoming network traffic at each location and updates the *nodeAffinity* rules. Finally, the Kubernetes scheduler reallocates the pods of the application according to the *nodeAffinity* rule.

Details regarding the algorithm used in ElasticFog are shown in Algorithm 1. Let *nodeList* be the list of Fog nodes in the cluster, while *applicationList* is the list of applications running in the cluster. Let *locationInfo* denote the list of nodes at each location (e.g., locationA: [node1]; locationB: [node2]), and let *appTrafficInfo* store information about the incoming network traffic of the application on each Fog node in Bps (e.g., node1: 800; node2: 200). Here, *weightInfo* denotes *weight* preference value for each location of the application in the *nodeAffinity* rule (e.g., locationA: 80; locationB: 20). *nodeList* and *applicationList* can be retrieved using the Kubernetes API. *locationInfo* can be updated based on the labels of the Fog nodes, while *appTrafficInfo* can be collected by reading the corresponding Endpoint object created for the application. The algorithm traverses all the applications running in the cluster. Based on *nodeList*, *locationInfo*, and *appTrafficInfo*, ElasticFog calculates the total incoming network traffic of the application at each location and in the cluster. Then, the proportion of incoming network traffic at each location is calculated, and the *weightInfo* is updated. The *weight* value of each location in the *nodeAffinity* rule is updated according to the proportional values in the *weightInfo*.

Once the *nodeAffinity* rule of the application is updated, the pods of the application are immediately rescheduled. To find the best-fit node to place the pod, the scheduler in Kubernetes carefully considers hard requirements, such as minimum requirement for CPU/RAM and health of the node, and soft requirements, such as balanced resource allocation, spreading the pods among nodes, and the node affinity preference that is updated by ElasticFog. Using the *weight* preference value in the *nodeAffinity* rule, the nodes belonging to the locations with a high proportion of incoming network traffic are prioritized when placing the pods.

It is interesting to note that ElasticFog also can exploit other schedulers to improve resource provisioning by considering multiple powerful metrics in the decision-making process. For example, paper [25] proposed a network-aware scheduler (NAS) which is an extension of the default scheduler in Kubernetes. After completing the *filtering* step and *scoring* step of the default scheduler in Kubernetes, NAS is used to consider the minimum bandwidth requirement and communication latency and find the best-fit node for the pod. NAS checks the minimum bandwidth requirement of each pod. If the node in the target location does not satisfy the minimum bandwidth requirement, NAS will find the location closest to the target location that satisfies the minimum bandwidth requirement to deploy the pod. For example, suppose that an application with nine replica pods is re-deployed in the cluster shown in Fig. 3, where each replica pod requires a minimum bandwidth. If location A has a large proportion of the incoming network traffic, 7/1/1 is the intended number of pods in locations A/B/C after the *filtering* and *scoring* step of the default Kubernetes scheduler. However, if the remaining bandwidth at location A is only enough to deploy 6 pods, the remaining pod will be deployed at location B (assuming that location B is closest to location A and its remaining band-

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3029583, IEEE Access

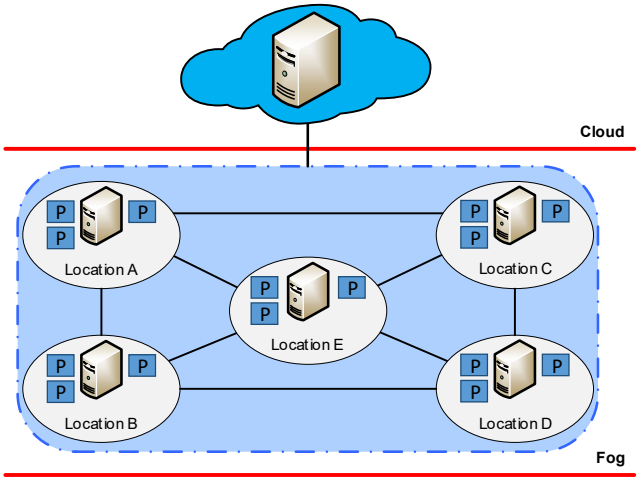

FIGURE 6. Medium-scale experiment of the Fog computing architecture.

width satisfies the minimum requirement). Consequently, the pod distribution in locations A/B/C become 6/2/1. Therefore, if NAS is implemented in the ElasticFog framework, multiple metrics related to the real-time network status (bandwidth, communication latency, and network traffic) can be fully considered to allocate appropriate resources for the application. In conclusion, ElasticFog can take into account diverse requirements and metrics about computation and network resources in real time to improve resource provisioning by exploiting different scheduling policies in Kubernetes.

V. PERFORMANCE EVALUATIONS

In this section, we evaluate the proposed ElasticFog framework in diverse situations. A cluster is set up including a set of several nodes with Kubernetes version 1.14.2 and Docker version 18.09.6. The master node runs with 4 CPU cores and 4 GB of RAM, and worker nodes, which are Fog nodes, run with 4 CPU cores and 3 GB of RAM. The evaluation is performed in an IoT infrastructure scenario in which geographically distributed Fog nodes provide services for IoT devices. The round-trip delay time between two locations is 10 ms. We use the *NodePort* Service to expose the application so that it is reachable from outside the cluster. If clients send requests to the application through wireless gateways linked with a Fog node at its location, the requests are transferred to the corresponding *NodePort* Service of the application on the Fog node. Apache HTTP server benchmarking tool (ab) [31] is used to create and send requests to the application.

We set up two experiments: a small-scale experiment (3 locations, 3 Fog nodes) and a medium-scale experiment (5 locations, 5 Fog nodes), as depicted in Fig. 3 and 6, respectively. Each location has one Fog node. The number of pod replicas of the application in the small-scale experiment is 9, while that in the medium-scale experiment is 15. We verified the operation and correctness of ElasticFog in the small-scale experiment. The efficiency of a proper pod distribution is evaluated in both experiments.

TABLE 1. Distribution of pods according to network traffic at each location.

Parameters	Network traffic (kBps)			Number of pods		
Location / Test case	A	B	C	A	B	C
1	12	0	0	7	1	1
2	8	4	0	6	2	1
3	7	5	0	5	3	1
4	6	6	0	4	4	1
5	4	4	4	3	3	3

A. DYNAMIC RESOURCE PROVISIONING USING ELASTICFOG

Table 1 illustrates the pod distribution according to the incoming network traffic in the small-scale experiment when using ElasticFog. In test case 1, when the incoming network traffic of the application on locations A, B, and C is 12, 0, and 0 kBps, respectively, the corresponding pod distribution at the three locations is 7, 1, and 1. Meanwhile, if there is 4 kBps of network traffic coming through locations A, B, and C, as in test case 5, the pods are evenly distributed to each location. By combining the real-time network information and the many beneficial policies for the application provided by the Kubernetes scheduler, the decision for pod distribution offers several advantages for the application as well as for the cluster. For instance, although there is no network traffic coming through locations B and C in test case 1, one pod is allocated to each location, which ensures the high availability of the application.

Fig. 7 depicts a dynamic adjustment of pod distribution in real time using ElasticFog. The amount of network traffic at each location fluctuates substantially during the experiment. The pod distribution is adjusted according to the proportion of incoming requests at each location. For example, when the proportion of incoming network traffic at the three locations is equal (at 0 s), the number of pods distributed to each of the three locations is the same: 3. When the proportion of incoming network traffic at location A rises to approximately 70% and the proportion of the other locations is only 15% each (at 450 s), the pod distribution becomes 6, 2, and 1 at locations A, B, and C, respectively. Similarly, when 100% of the incoming network traffic is at location A (at 900 s), the pod distribution becomes 7, 1, and 1 to A, B, and C, respectively. This proves that ElasticFog is aware of the changes to the network traffic status in real time and quickly adjusts the number of pods among locations.

Because the application can be frequently re-scheduled when using ElasticFog, it is worth evaluating the total time for the deployment and re-deployment process, as shown in Fig. 8. The deployment time is measured from the application is deployed until all of its replica pods become "running" status (ready to serve the client requests). The re-deployment time is measured from the application is re-scheduled until all the replica pods return to "running" status. The re-deployment time is 11.5 s in the case of the small-scale experiment and 17 s in the case of the medium-scale experiment. In both experiments, the re-deployment time

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3029583, IEEE Access

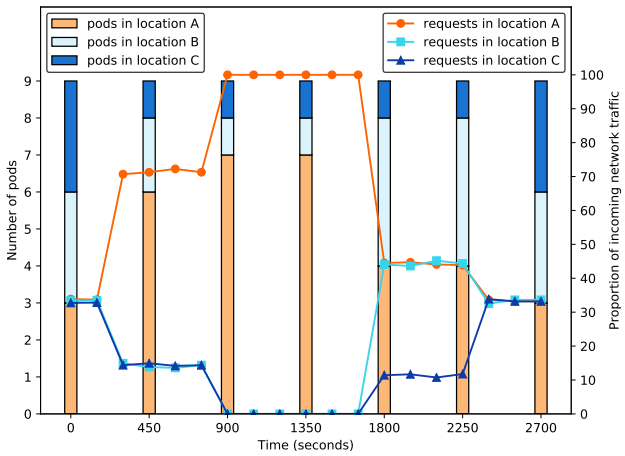

FIGURE 7. Dynamic adjustment of pod distribution in real time.

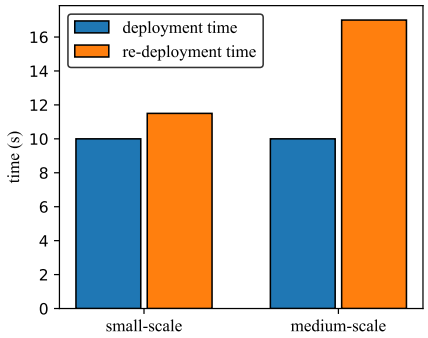

FIGURE 8. Total time for deployment and re-deployment process.

is higher than the deployment time. This is because the re-deployment process requires Kubernetes to kill the old pod and create a new one sequentially. It ensures that the application always has a certain number of available pods to handle client requests during the re-deployment process, to ensure the high availability of the application.

B. EFFECT OF POD DISTRIBUTION IN THE FOG COMPUTING ENVIRONMENT

To analyze the effect of pod distribution on system performance in the Fog computing environment, we evaluate the throughput and latency between different pod distributions as shown in Figs. 9 and 10. Scenario 1 evenly distributes pods to each location, while scenario 2 has pods concentrated on location A. For example, the pod distributions in the small and medium-scale experiments are 3, 3, 3 pods and 3, 3, 3, 3, 3 pods, respectively, in scenario 1, while they are 7, 1, 1 and 11, 1, 1, 1, 1 pods in scenario 2. In this evaluation, we focus on the network traffic accessing one location (location A). We increase the incoming network traffic at location A by increasing the number of concurrent requests having the same size from 1 to 16.

To balance the workload among the Fog nodes in Kubernetes, the requests may be redirected to pods belonging to Fog nodes at other locations. If many requests are redirected to other locations, it may significantly increase the latency, due to the network delay between locations. As illustrated in Fig. 9(a) and 10(a), the median latency in scenario 1 is much higher than that in scenario 2, and it tends to increase with an increase in the number of concurrent requests. Consequently, the throughput in scenario 1 is significantly lower than that in scenario 2, as shown in Fig. 9(b) and 10(b). In the small-scale experiment, the throughput obtained with 1 request in scenario 2 is approximately 140% larger than that obtained in scenario 1, and the throughput in scenario 2 is twice higher than that in scenario 1 as the number of concurrent requests increases. Because location A consists of three pods in scenario 1 and seven pods in scenario 2, the proportion of requests handled at location A in scenario 1 is significantly lower than that in scenario 2. In other words, most requests in scenario 2 can be handled immediately at location A, whereas a large proportion of the requests in scenario 1 are considerably delayed by redirection to other locations. Likewise, the throughput in scenario 2 of the medium-scale experiment, which has 11 out of 15 pods at location A, is twice higher compared with scenario 1.

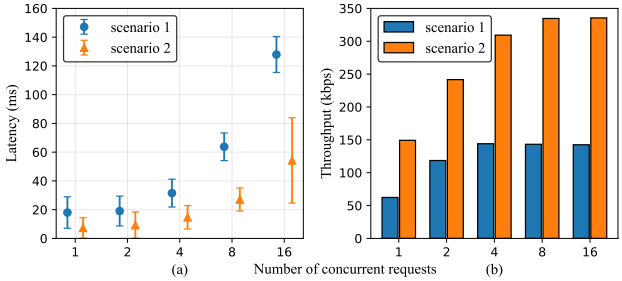

FIGURE 9. Performance at location A with different pod distributions in the small-scale experiment: (a) Latency; (b) Throughput.

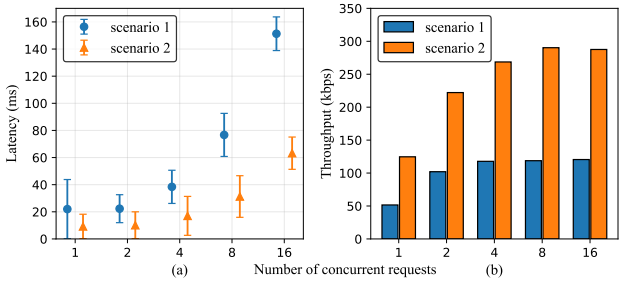

FIGURE 10. Performance at location A with different pod distributions in the medium-scale experiment: (a) Latency; (b) Throughput.

Figs. 11 - 14 compare the latency and cumulative throughput between ElasticFog and the default mechanism in Kubernetes with network traffic variation among locations. The network traffic at each location is changed by sending a different number of concurrent requests that have the same size to the location. Note that the numbers in the format

This article has been accepted for publication in a future issue of this journal, but has not been fully edited. Content may change prior to final publication. Citation information: DOI 10.1109/ACCESS.2020.3029583, IEEE Access

TABLE 2. Pod distribution according to the ratio of concurrent requests in the small-scale experiment

Ratio of concurrent requests (A:B:C)	Number of pods (A,B,C)
2:2:2	3, 3, 3
4:4:1	4, 4, 1
6:2:1	6, 2, 1
8:1:1	7, 1, 1

TABLE 3. Pod distribution according to the ratio of concurrent requests in the medium-scale experiment

Ratio of concurrent requests (A:B:C:D:E)	Number of pods (A,B,C,D,E)
2:2:2:2:2	3, 3, 3, 3, 3
4:4:1:1:1	6, 6, 1, 1, 1
6:2:1:1:1	10, 2, 1, 1, 1
8:1:1:1:1	11, 1, 1, 1, 1

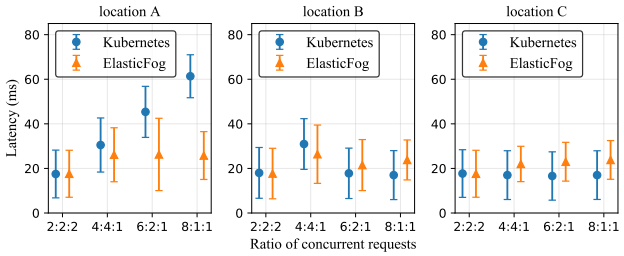

FIGURE 11. Latency of requests at three locations in the small-scale experiment.

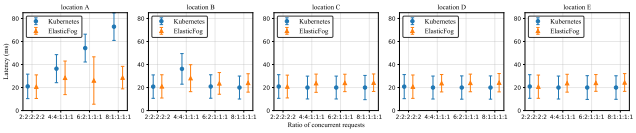

FIGURE 12. Latency of requests at five locations in the medium-scale experiment.

a:b:c indicate the number of concurrent requests at each of locations A, B, and C, respectively, in the case of the small-scale experiment. In the medium-scale experiment, a:b:c:d:e represents the number of requests accessing locations A, B, C, D, and E, respectively. We assume that the default Kubernetes mechanism always maintains an even number of pods among locations (3 pods on each location) regardless of changes in the network traffic status, while ElasticFog dynamically adjusts the number of pods at each location according to the ratio of concurrent requests. The number of concurrent requests and the corresponding number of pods in the ElasticFog case for the small-scale and medium-scale experiments are shown in Table 2 and 3.

As observed in Figs. 11 and 12, the median latency for requests at location A in the default Kubernetes mechanism tends to increase from 18 ms to 61 ms (in the small-scale experiment) and from 21 ms to 73 ms (in the medium-scale experiment) as the number of concurrent requests at this location increases from 2 to 8. In contrast, it varies from 18 ms to approximately 25 ms (in the small-scale experiment) and from 21 ms to approximately 28 ms (in the medium-scale experiment) in the case of ElasticFog. This means that the increase in the number of pods at location A effectively minimizes the latency for handling the client requests when there is an increase in the at this location. It is important to note that when the Fog node at location A has more pods, the number of pods in other nodes located at locations B and C will decrease, because the number of replica pods of the application is fixed. Therefore, the latency of the requests at locations B and C in ElasticFog can, in some cases, be slightly higher than that in the default Kubernetes mechanism.

Figs. 13 and 14 illustrate the cumulative throughput of the clients on all locations. In the default Kubernetes mechanism, the cumulative throughput for the 8:1:1 case is decreased approximately 26% compared to the 2:2:2 case in the small-scale experiment, and it shows more than 35% degradation compared between the 2:2:2:2:2 and 8:1:1:1:1 case in the medium-scale experiment. This is because the number of pods at location A does not change in the default Kubernetes even if the network traffic coming to that location increases significantly, as proved by the fact that the throughput at location A does not increase. Meanwhile, the throughput at location A in ElasticFog tends to increase because the number of pods in this location is increased according to the increase in the incoming network traffic. In some cases, the number of pods in the other locations of the default Kubernetes may be higher than that in ElasticFog. For example, in the small-scale experiment, the number of pods in locations B and C in the Kubernetes mechanism is higher than that in ElasticFog in the case of 8:1:1. However, the throughput at these locations of the default Kubernetes is just slightly higher than that in ElasticFog. This is because the amount of network traffic coming through these locations is not high, and a small number of pods can still handle the requests effectively. Consequently, the cumulative throughput at the three locations of ElasticFog is approximately 59% higher than the Kubernetes mechanism in the 8:1:1 case. It is important to note that Kubernetes tends to face a decrease in cumulative throughput with changes in network traffic, whereas ElasticFog maintains the cumulative throughput because of dynamic reallocation of the resources according to the distribution of network traffic at each location.

The efficiency of ElasticFog can also be seen clearly in the medium-scale experiment. For example, the cumulative throughput of five locations of ElasticFog is higher by approximately 10% in the 4:1:1:1:1 case and by approximately 44% in the 8:1:1:1:1 case when compared with Kubernetes. Overall, we can conclude that the cumulative throughput in the Kubernetes mechanism worsens for unbalanced incoming network traffic because of the lack of dynamic adjustment of the pod distribution, while ElasticFog maintains a high cumulative throughput regardless of the distribution of network

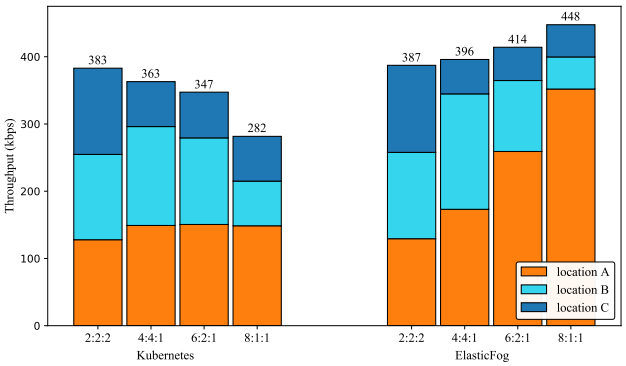

FIGURE 13. Cumulative throughput of requests at three locations in the small-scale experiment.

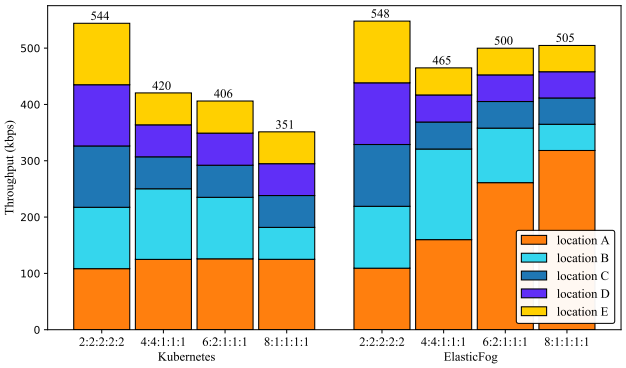

FIGURE 14. Cumulative throughput of requests at five locations in the medium-scale experiment.

traffic. Therefore, it is important to distribute the pods of an application appropriately among locations to minimize the latency and maximize the throughput of the client requests in the cluster.

VI. CONCLUSIONS

Fog computing and container-based applications are emerging as solutions to deal with the challenges of a tremendous number of IoT services. The Kubernetes platform, which provides powerful and flexible features, has been exploited for the management of containerized applications in Fog computing. In this paper, we proposed a real-time elastic resource provisioning for applications in container-based Fog computing. ElasticFog is implemented based on the Kubernetes platform, and it collects the network traffic status to provide elastic resource provisioning of the application among geographically distributed Fog nodes in real time. By combining the network traffic information and the Kubernetes scheduler, we can consider not only the real-time network traffic status of the application on each Fog node but also various useful policies supported by the Kubernetes scheduler to take decisions on resource provisioning for the application. The experimental results with the small-scale and medium-scale experiments proved that ElasticFog significantly improves the system performance in terms of throughput and latency compared with the default mechanism in Kubernetes. Thus, it is important to provide a dynamic resource provisioning based on changes in network traffic status in real time to minimize latency and maximize the throughput of client requests in the Fog computing environment.

REFERENCES

[1] Cisco, "Cisco annual internet report (2018–2023) white paper." [Online]. Available: https://www.cisco.com/c/en/us/solutions/collateral/executive-perspectives/annual-internet-report/white-paper-c11-741490.html.

[2] Q. Zhang, L. Cheng, and R. Boutaba, "Cloud computing: state-of-the-art and research challenges," *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[3] X. Sun and N. Ansari, "Edgeiot: Mobile edge computing for the internet of things," *IEEE Communications Magazine*, vol. 54, no. 12, pp. 22–29, 2016.

[4] N. Mohamed, J. Al-Jaroodi, I. Jawhar, S. Lazarova-Molnar, and S. Mahmoud, "Smartcityware: A service-oriented middleware for cloud and fog enabled smart city services," *IEEE Access*, vol. 5, pp. 17 576–17 588, 2017.

[5] M. Mukherjee, L. Shu, and D. Wang, "Survey of fog computing: Fundamental, network applications, and research challenges," *IEEE Communications Surveys Tutorials*, vol. 20, no. 3, pp. 1826–1857, 2018.

[6] M. Chiang and T. Zhang, "Fog and iot: An overview of research opportunities," *IEEE Internet of Things Journal*, vol. 3, no. 6, pp. 854–864, 2016.

[7] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, 2012, pp. 13–16.

[8] A. V. Dastjerdi and R. Buyya, "Fog computing: Helping the internet of things realize its potential," *Computer*, vol. 49, no. 8, pp. 112–116, 2016.

[9] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE international symposium on performance analysis of systems and software (ISPASS)*. IEEE, 2015, pp. 171–172.

[10] J. Luo, L. Yin, J. Hu, C. Wang, X. Liu, X. Fan, and H. Luo, "Container-based fog computing architecture and energy-balancing scheduling algorithm for energy iot," *Future Generation Computer Systems*, vol. 97, pp. 50–60, 2019.

[11] R. K. Naha, S. Garg, D. Georgakopoulos, P. P. Jayaraman, L. Gao, Y. Xiang, and R. Ranjan, "Fog computing: Survey of trends, architectures, requirements, and research directions," *IEEE access*, vol. 6, pp. 47 980–48 009, 2018.

[12] Kubernetes, "Kubernetes, production-grade container orchestration." [Online]. Available: https://kubernetes.io/

[13] D. Roca, J. V. Quiroga, M. Valero, and M. Nemirovsky, "Fog function virtualization: A flexible solution for iot applications," in *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*. IEEE, 2017, pp. 74–80.

[14] O. Skarlat, S. Schulte, M. Borkowski, and P. Leitner, "Resource provisioning for iot services in the fog," in *2016 IEEE 9th international conference on service-oriented computing and applications (SOCA)*. IEEE, 2016, pp. 32–39.

[15] D. Zeng, L. Gu, S. Guo, Z. Cheng, and S. Yu, "Joint optimization of task scheduling and image placement in fog computing supported software-defined embedded system," *IEEE Transactions on Computers*, vol. 65, no. 12, pp. 3702–3712, 2016.

[16] M. Aazam and E.-N. Huh, "Dynamic resource provisioning through fog micro datacenter," in *2015 IEEE international conference on pervasive computing and communication workshops (PerCom workshops)*. IEEE, 2015, pp. 105–110.

[17] L. Ni, J. Zhang, C. Jiang, C. Yan, and K. Yu, "Resource allocation strategy in fog computing based on priced timed petri nets," *IEEE Internet of Things Journal*, vol. 4, no. 5, pp. 1216–1228, 2017.

[18] M. Abbasi, M. Yaghoobikia, M. Rafiee, A. Jolfaei, and M. R. Khosravi, "Efficient resource management and workload allocation in fog–cloud computing paradigm in iot using learning classifier systems," *Computer Communications*, vol. 153, pp. 217–228, 2020.

[19] S. Hoque, M. S. de Brito, A. Willner, O. Keil, and T. Magedanz, "Towards container orchestration in fog computing infrastructures," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2. IEEE, 2017, pp. 294–299.

[20] D. Santoro, D. Zozin, D. Pizzolli, F. De Pellegrini, and S. Cretti, "Foggy: a platform for workload orchestration in a fog computing environment," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, 2017, pp. 231–234.
[21] C.-H. Hong, K. Lee, M. Kang, and C. Yoo, "qcon: Qos-aware network resource management for fog computing," *Sensors*, vol. 18, no. 10, p. 3444, 2018.
[22] D. Zhao, M. Mohamed, and H. Ludwig, "Locality-aware scheduling for containers in cloud computing," *IEEE Transactions on Cloud Computing*, 2018.
[23] L. Yin, J. Luo, and H. Luo, "Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4712–4721, 2018.
[24] C. Wöbker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: Deployment and management of fog computing applications," in *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018, pp. 1–7.
[25] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *2019 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2019, pp. 351–359.
[26] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Resource provisioning in fog computing: From theory to practice," *Sensors*, vol. 19, no. 10, p. 2238, 2019.
[27] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards delay-aware container-based service function chaining in fog computing," in *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2020, pp. 1–9.
[28] F. Rossi, V. Cardellini, F. L. Presti, and M. Nardelli, "Geo-distributed efficient deployment of containers with kubernetes," *Computer Communications*, 2020.
[29] Docker, "Docker, empowering app development for developers." [Online]. Available: https://www.docker.com/
[30] S. Sarkar, S. Chatterjee, and S. Misra, "Assessment of the suitability of fog computing in the context of internet of things," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 46–59, 2015.
[31] ab, "ab - apache http server benchmarking tool." [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html

NGUYEN DINH NGUYEN received his B.S. degree in electronics and telecommunications from Vietnam National University, Hanoi in 2017. Now he is pursuing his Master degree in the School of Information and Communication Engineering, Chungbuk National University. His research interests cover cloud computing, edge computing, SDN/NFV, and Internet of Things.

LINH-AN PHAN received the B.S. degree in information technology from the University of Science and Technology, University of Da Nang, Vietnam, in 2013, and the M.S. degree in information and communication engineering from Chungbuk National University, Korea, in 2019. He worked as a Software Engineer at Samsung Vietnam Mobile R&D Center, Vietnam from 2013 to 2017. Since 2019, he has been a PhD Student with the School of Information and Communication Engineering, Chungbuk National University, Korea. His research interests include wireless networks, the Internet of Things, and cloud/edge computing.

DAE-HEON PARK received B.S, M.S., and Ph.D degrees in Communication & Information engineering from Sunchon National University in 2006, 2008, and 2015. Since 2011, he has been a senior research at ETRI, Korea. His research interests are IoT, AI, Cloud, BigData, and ICT convergence with agriculture.

SEHAN KIM received the B.S and M.S degrees in Computer Engineering from Korea Aerospace University, Korea, in 1998 and 2000. He worked as a research staff at Samsung Advanced Institute of Technology in 2000. Since 2001, he has been a Principal Researcher, Director at ETRI, Korea. His research interests are Digital Twin, Platform, Data Science with IoT, AI, Cloud, BigData, and intelligent ICT convergence with Agriculture, food & Fisheries.

TAEHONG KIM received his B.S. degree in computer science from Ajou University, Korea, in 2005, and his M.S. degree in information and communication engineering from Korea Advanced Institute of Science and Technology (KAIST) in 2007. He received his Ph.D. degree in computer science from KAIST in 2012. He worked as a research staff member at Samsung Advanced Institute of Technology (SAIT) and Samsung DMC R&D Center from 2012 to 2014. He also worked as a senior researcher at the Electronics and Telecommunications Research Institute (ETRI), Korea, from 2014 to 2016. Since 2016, he has been an associate professor with the School of Information and Communication Engineering, Chungbuk National University, Korea. He has been an associate editor of IEEE Access since 2020. His research interests include edge computing, SDN/NFV, the Internet of Things, and wireless sensor networks.