IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received January 23, 2022, accepted February 7, 2022, date of publication February 10, 2022, date of current version February 22, 2022.

Digital Object Identifier 10.1109/ACCESS.2022.3150867

Traffic-Aware Horizontal Pod Autoscaler in Kubernetes-Based Edge Computing Infrastructure

LE HOANG PHUC, LINH-AN PHAN, (Associate Member, IEEE), AND TAEHONG KIM, (Member, IEEE)

School of Information and Communication Engineering, Chungbuk National University, Cheongju 28644, South Korea

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

This work was supported by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (No. 2021R1F1A1060328).

ABSTRACT Container-based Internet of Things (IoT) applications in an edge computing environment require autoscaling to dynamically adapt to fluctuations in IoT device requests. Although Kubernetes' horizontal pod autoscaler provides the resource autoscaling feature by monitoring the resource status of nodes and then making pod adjustments if necessary, it evenly allocates pods to worker nodes without considering the imbalance of resource demand between nodes in an edge computing environment. This paper proposes the traffic-aware horizontal pod autoscaler (THPA), which operates on top of Kubernetes to enable real-time traffic-aware resource autoscaling for IoT applications in an edge computing environment. THPA performs upscaling and downscaling actions based on network traffic information from nodes to improve the quality of IoT services in the edge computing infrastructure. Experimental results show that Kubernetes with THPA improves the average response time and throughput of IoT applications by approximately 150% compared to Kubernetes with the horizontal pod autoscaler. This indicates that it is important to provide proper resource scaling according to the network traffic distribution to maximize IoT applications performance in an edge computing environment.

INDEX TERMS Kubernetes, horizontal pod autoscaler, network-aware resource provisioning, IoT.

I. INTRODUCTION

With the massive boom in Internet of Things (IoT) applications in daily life, such as in industry, healthcare, logistics, and military [1], [2], many IoT devices collect data from the environment and generate huge amounts of data. These data are transferred to and processed at cloud servers located at the network core, resulting in high bandwidth consumption and response time [3]. Recently, many studies have aimed to improve the quality of IoT services in cloud computing, e.g., [4], [5]. However, in particular, the response time of cloud computing has become unable to meet the requirements of many time-sensitive IoT services, such as augmented/virtual reality, smart factories, and smart transportation systems [6], [7]. Edge computing is a new paradigm that overcomes the inherent limitations of cloud computing by distributing edge nodes with computing resources closer to IoT devices [8], [9]. By processing data at local edge nodes without transferring them to the cloud, edge computing can sufficiently reduce the response time to satisfy the requirements of time-critical applications [10].

Using containerization, due to the lightness and portability of containers [11], it is easy to deploy, install, update, and delete application services on edge nodes, and various types of IoT services can be provided simultaneously at each edge node. As such, containerization is widely considered as the most suitable technology for providing IoT services in edge computing environments. However, containerization technology is limited to deploying and managing container-level application services, requiring container orchestration to monitor and manage resource status through multiple edge nodes in an edge computing environment [12], [13].

Kubernetes [14] is a representative open source-based container orchestration platform that provides a variety of functions, such as service placement, resource monitoring of edge nodes, and service automation. For example, Kubernetes can easily deploy container-based IoT service to designated

The associate editor coordinating the review of this manuscript and approving it for publication was Mehdi Hosseinzadeh.

edge nodes in the form of pod without requiring any manual configuration. Moreover, Kubernetes autoscaling can optimize the resource usage and cost by automatically upscaling or downscaling resources according to demand. Among the several autoscaling options provided by Kubernetes, such as cluster autoscaler, vertical pod autoscaler, and horizontal pod autoscaler (HPA), the HPA provides seamless autoscaling by dynamically adjusting the number of pods without restarting the existing pods [15]; thus, it can play a key role in an edge computing environment that requires both high availability and dynamic resource adjustment.

However, despite the many benefits of Kubernetes, it is still in its infancy in an edge computing environment. In edge computing infrastructure, requests from devices are handled by container-based applications on edge nodes, and the traffic load varies over location and time. Namely, as some nodes are too busy to handle a large amount of traffic while others are idle [16], an imbalance of demand occurs between nodes. In Kubernetes, the kube-proxy balances resource usage between nodes by sharing the incoming traffic at each node to all pods in the cluster in a random or round-robin manner. Because edge nodes are geographically distributed, there is network delay in their communication, so this kind of redirection offered by the kube-proxy can increase the response time of applications. Therefore, it is necessary to allocate more or terminate redundant computational resources according to network traffic at each node to maximize the amount of locally handled traffic while minimizing network delay by minimizing the number of requests handled by pods on remote nodes. Nevertheless, when the resource demands of the applications change, Kubernetes' HPA (KHPA) only tries to evenly distribute new pods to nodes or terminate redundant pods on nodes based on pod status without considering the network delay between edge nodes and the volume of network traffic accessing them in real time. This limitation of KHPA can result in the degradation of the quality of service and overall throughput of the system [17].

To solve the aforementioned problem of KHPA in a Kubernetes-based edge computing infrastructure, this paper proposes traffic-aware HPA (THPA), which operates on top of Kubernetes to provide dynamic resource autoscaling by considering the IoT service demand at each edge node. Specifically, in an upscaling event, THPA allocates a number of additional pods proportional to the distribution of network traffic accessing the nodes, whereas in a downscaling event, it terminates the pods in the node with low demand. Experimental evaluations prove that THPA significantly improves the average response time and throughput by maximizing the amount of traffic handled locally and avoiding the round-trip delay from redirection between edge nodes in an edge computing environment.

The remainder of this paper is organized as follows. Section II presents related works, and Section III describes the fundamental background about Kubernetes. Section IV describes the proposed THPA and how it solves the problem of KHPA in an edge computing environment. Performance evaluations of THPA compared with KHPA in various traffic scenarios are reported in Section V. Finally, we conclude the paper in Section VI.

II. RELATED WORKS

Since its first announcement by Google in 2014 [18], Kubernetes has been the most prominent container orchestration platform for many applications that require the ease of deployment, scaling, and management. Although new features have been continuously released to improve application production environments, there have been many studies addressing the limitations of resource provisioning and scalability in Kubernetes [19], including serveral in recent years.

Santos *et al.* [20], [21] proposed the network-aware scheduler (NAS), which is an effective scheduling mechanism implemented by extending Kubernetes' scheduler to reduce latency and prevent bandwidth usage violations for container-based IoT applications. The NAS selects a node to schedule a pod by finding the best node that satisfies the bandwidth constraint and round-trip time to the targeted node. Likewise, in [22], the service function chaining (SFC) controller was proposed to optimize the resource provisioning in the Kubernetes-based fog computing environment. SFC was developed by extending the Kubernetes scheduler to enable latency-aware or location-aware resource scheduling to improve the quality of smart city applications. However, in [20], [21], and [22], latency was built in to the initial configurations, so these approaches can not provide dynamic adaption for latency and throughput varying in time.

NetMARKS [23] was proposed as a scheduler extender to enable traffic-aware resource provisioning. Specifically, NetMARKS improves Kubernetes scheduling by exploiting dynamic network metrics collected with the Istio Service Mesh. Similarly, ElasticFog [17] enables elastic resource provisioning in a fog computing environment. The aim of the corresponding study was to appropriately allocate resources for each location based on its real-time incoming network traffic to improve the quality of IoT services. Nevertheless, the quality of service can not be guaranteed in the presence of computational resources overhead since the approaches of [23] and [17] do not consider the amount of network traffic accessing the system, nor do they support autoscaling.

Libra [24] is a hybrid scaling mechanism implemented on Kubernetes by mixing both vertical and horizontal scaling mechanisms. The aim of Libra is to control the horizontal scaling process by finding and updating the appropriate resource limit for each application pod. Interestingly, many studies have employed machine learning to solve the scalability problem. Tengfei Hu *et al.* [25] proposed an intelligent enhancement of KHPA by applying a forecast model to predict the number of replicas to be adjusted in a cluster. This approach improves the automation of HPA and optimizes the application performance based on the load fluctuation. L.Toka *et al.* [26] proposed HPA+, which provides proactive autoscaling to improve the quality of application services

by exploiting the multi-forecast machine learning models. HPA+ applies the best prediction result from these forecasting models as the custom metric if their accuracy proves to equal that of the custom metric; otherwise, HPA+ uses CPU metrics to make scaling decisions. However, despite efforts such as mixing both vertical and horizontal scaling mechanisms [24] and applying the machine learning to KHPA [25], [26], the quality of application services can be reduced if the network delay between worker nodes in an edge computing environment is high.

Although the aforementioned studies have introduced many approaches to address the ongoing problems of resource provisioning and scalability, their proposed approaches do not address the scalability problem of Kubernetes in an edge computing environment, where the amount of data fluctuates over time as well as location. In this paper, we propose THPA to adapt to the demands of applications in an edge computing environment, and we prove that the quality of service can be significantly improved by both upscaling and downscaling pods based on the network traffic information at each edge node.

III. PRELIMINARIES

This section introduces the fundamentals of the Kubernetes concept, its scheduling, and KHPA mechanisms to better describe our proposed mechanism.

A. KUBERNETES

Kubernetes is a prominent open-source platform that automates the process of deploying, managing and scaling containerized applications. A Kubernetes cluster consists of master nodes and worker nodes, and the running containerized applications are managed in a unit called a pod, which is the smallest execution unit in Kubernetes.

The control plane of a cluster is responsible for monitoring and managing worker nodes and pods and consists of the kube-apiserver, etcd, kube-controller-manager, and kube-scheduler. The kube-apiserver exposes the Kubernetes API and communicates with node components. The system administrator can interact and control the system via the command-line tool of the kube-apiserver. In Kubernetes, data such as metadata, current state, and desired state of all Kubernetes' resources are stored in the etcd under key-value format [27]. The kube-controller-manager is a core control loop in Kubernetes, and it assures the matching between the current state and desired state of a cluster. The kube-controller-manager continuously watches for the current state of the cluster via the kube-apiserver and it updates, creates, and terminates resources if necessary [28]. The creation of a pod is managed by the kube-scheduler, which finds the best node for an unscheduled pod based on the pod's container specification, scheduling policies, and available resources across nodes [29].

A worker node is composed of the kube-proxy, kubelet and container runtime, which runs on every node in a cluster to provide a runtime environment and keep pods healthy. The kube-proxy is a network proxy that runs on each node and maintains network rules that are necessary for establishing the connection to pods from inside or outside the cluster. Kubelet ensures that pods on a node are running and healthy, and the container runtime (e.g.Docker) enables the execution of containers and management of container images on a node [30].

In a Kubernetes cluster, it is difficult to rely on the IP address of an application pod to access the application because the pod's IP changes whenever it is restarted. To ensure the reachability of an application, an abstraction layer called a Service is used to expose a group of application pods to the clients. A virtual IP address called ClusterIP is assigned to each Service. Because ClusterIP is not changed unless it is re-created, an application's reachability can be guaranteed [31]. Note that Cluster IP is only reachable within a cluster. If a Service needs to be reachable from outside of a cluster, it must be configured with NodePort or LoadBalancer. Whenever a NodePort Service is created, a static port is opened on each node and the application can be accessed from inside or outside of the cluster using nodeIP:port. LoadBalancer is another option that uses the load balancing mechanisms provided by the cloud provider to expose the Service externally. Whenever there is traffic accessing an application, it is routed to the application pods, which are watched by the Service. The routing decision is based on kube-proxy modes, such as user-space, iptables, and IPVS. By default, the kube-proxy works in user-space mode, which chooses a pod to handle the traffic based on a round-robin algorithm. Different from user-space mode, iptables mode randomly picks out a pod for traffic handling. For large scale applications, IPVS provides much more efficient routing algorithms, such as round-robin, least connection, destination hashing, source hashing, shortest expected delay, and never queue [32].

B. KUBERNETES SCHEDULING MECHANISM

In Kubernetes, scheduling is a process of choosing the best node to place a new pod, which is performed by the kube-scheduler on the control plane. The kube-scheduler's node selection process follows two steps: filtering and scoring. The filtering step finds feasible nodes that satisfy the predefined predicates in the kube-scheduler configuration, such as *PodFitsResources, PodFitsHostPorts, NoDiskConflict, etc* [33]. Based on the set of candidate nodes from the filtering step, the scoring step assigns a score to nodes according to priorities, such as *SelectorSpreadPriority, LeastRequestedPriority, MostRequestedPriority, etc* [33]. After the scoring step, the final scheduling decision can be made by selecting the node with the highest score, and the pod is scheduled and run on the selected node.

C. KUBERNETES HORIZONTAL POD AUTOSCALER

The HPA is a Kubernetes's controller that periodically adjusts the number of pod replicas to move the current state of a

cluster towards the desired state based on targeted metric values [34]. These metrics can be set based on the average usage of resources such as CPU, memory, or a combination of them. Moreover, Kubernetes supports custom metrics to support various demands of applications.

Algorithm 1 KHPA Algorithm

pods : *list of application pods in cluster*.
curPods : *current number of application pods*.
dPods : *desired number of application pods*.
curMetricVal : *current metric value*.
dMetricVal : *desired metric value*.
HPA_Sync_Period : *HPA sync period*.

```
1: while true do
2:     curPods = getCurPods(app)
3:     curMetricVal = getCurMetricValue(app)
4:     dMetricVal = getDesiredMetricValue(app)
5:     ratio = curMetricVal ÷ dMetricVal
6:     dPods = ceil[ratio × curPods]
7:     if dPods ! = curPods then
8:         setDesiredPods(app, dPods)
9:     end if
10:    time.sleep(HPA_Sync_Period)
11: end while
```

Algorithm 1 describes the detailed algorithm of KHPA. Let *curPods* and *dPods* denote the current and desired numbers of application pods, respectively, while *curMetricVal* and *dMetricVal* store the current and desired metric values, respectively. In each period defined by *HPA_Sync_Period*, KHPA collects the current number of pods in the cluster (*curPods*), current metric value (*curMetricVal*), and desired metric value (*dMetricVal*) to calculate the desired number of pods (*dPods*). If the desired number of pods is different from the current number of pods in the cluster, KHPA adjusts the number of pods in the cluster to the desired number (line 8 in Algorithm 1).

If the desired number of application pods is higher than the current number of pods in the cluster, the upscaling process is invoked, whereas in the opposite case, the downscale process is invoked. For example, assuming that the CPU utilization metric is used, if the current metric value is 100 m (where m stands for millicore) and the desired metric value is 50 m, KHPA doubles the number of pods in the cluster. In contrast, when the current metric value is 50 m and the desired metric value is 100 m, KHPA halves the current number of pods in the cluster. Note that if the current metric value is equal to the desired metric value, the number of application pods in the cluster remains the same.

IV. TRAFFIC-AWARE HORIZONTAL POD AUTOSCALER

In this section, we discuss the Kubernetes based edge computing infrastructure and describe the current problem faced by KHPA. Then, we describe THPA, which automatically scales the number of application pods up or down according to real-time traffic information in an edge computing environment.

A. KUBERNETES-BASED EDGE COMPUTING INFRASTRUCTURE

Fig. 1 (a) illustrates the architecture of a Kubernetes-based edge computing system. IoT devices can access the application services being deployed on edge nodes in the form of pods, and there is a considerable network delay between node communication. Note that the kube-proxy is configured with the round-robin load balancing algorithm which evenly distributes incoming requests at each worker node to all pods in the cluster to balance the resource usage among nodes. Furthermore, to adapt to the application demand, which varies in time, KHPA is enabled to periodically inspect the current and desired CPU metric values to determine whenever the number of pods in a cluster need to be adjusted [19].

Although the incoming traffic is different among locations (edge nodes), which indicates that their resource demands are not identical, KHPA only tries to evenly allocate additional pods or terminate redundant pods in a cluster whenever the resource demand is above or below the threshold. As such, a proportion of incoming requests at a specific node are locally handled by local pods, while the remainder are distributed to pods on remote nodes by the kube-proxy, and this kind of redirection with the existence of network delay between nodes significantly reduces the application quality [35]. More specifically, without awareness of the incoming traffic at each node, the adjustment process of KHPA can unwittingly increase the proportion of incoming traffic that is redirected to remote nodes and therefore increase the response time of applications at a specific node. For example, in Fig. 1 (a), assuming that each device sends one request to an application on each node at a time, KHPA increases the number of application pods in the cluster from 3 to 6 as the network traffic accessing worker 1 increases from 1 to 7. Then, without awareness of the incoming requests at each node, KHPA simply evenly distributes new pods to the worker nodes, which cause the numbers of pods on workers 1, 2, and 3 become 2, 2, and 2, respectively. It is clear to see that only 1/3 of the incoming requests at worker 1 can be locally handled because worker 1 has 2 application pods, compared with the total 6 application pods in the cluster. Meanwhile, 2/3 of the incoming requests at worker 1 are distributed to other pods on remote nodes by the kube-proxy for processing, which indicates that these requests require more time to response. Therefore, the application performance can not be improved significantly although more resources are put into the cluster.

Considering the aforementioned problem, we propose THPA, which adjusts application pods on edge nodes based on the proportion of incoming traffic accessing nodes in real time during the scaling process. For example, in Fig. 1 (b), assuming that THPA decides to add three more application pods to the cluster (upscaling) as the number of requests

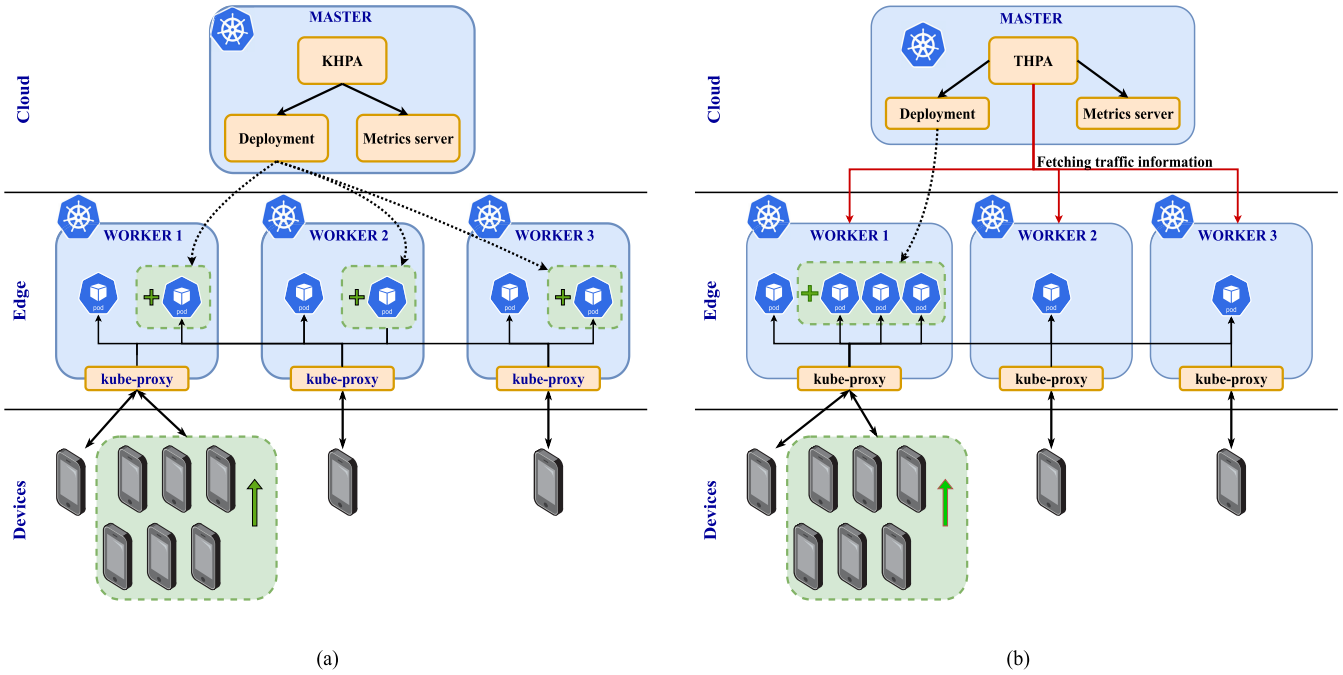

FIGURE 1. (a) KHPA in Kubernetes-based Edge computing Architecture and (b) THPA in Kubernetes-based Edge computing Architecture.

accessing worker 1 increases from 1 to 7. Before distributing new pods to the cluster, THPA collects the network traffic information at each worker node, which results in 7:1:1 distribution, where w1:w2:w3 denotes the number of concurrent requests accessing worker 1, 2, and 3, respectively. In addition, w1-w2-w3 represents the number of application pods on worker 1, 2, and 3, respectively, and it can also be used to indicate the number of pods to be added or deleted. Then, THPA calculates the number of new pods to allocate to each worker node based on the collected traffic information, which results in 3-0-0. This result indicates that all new pods are distributed to worker 1, while the numbers of pods on other nodes remain the same. After finishing the upscaling process, the pod distribution in the cluster becomes 4-1-1. From this result, it is clear to see that 2/3 of the incoming requests at worker 1 can be handled locally, whereas only 1/3 of them are redirected to remote nodes for processing, which significantly reduces the response time of the application at worker 1. Therefore, by adjusting the number of pods in the cluster according to the traffic distribution, THPA can minimize the application response time compared to KHPA. It is noteworthy that THPA also conducts downscaling, where it terminates application pods on edge nodes based on the network traffic accessing them in real time.

B. TRAFFIC-AWARE HORIZONTAL POD AUTOSCALER

This subsection describes the detailed THPA algorithm, which consists of Algorithms 2, 3, and 4.

In Kubernetes, it is important to note that KHPA itself does not allocate or terminate pods during upscaling or downscaling; it only updates the desired number of application pods. Then, after the desired number of pods is updated, in the case of upscaling, the kube-scheduler schedules new pods to worker nodes, whereas the Replicaset controller terminates redundant application pods when downscaling occurs. Therefore, to make both the upscaling and downscaling processes aware of network traffic, THPA modifies the pod scheduling and terminating processes as described in Algorithm 3 and 4, respectively.

Moreover, because THPA requires real-time traffic information at worker nodes in a Kubernetes cluster (lines 9 and 17 in Algorithm 2), we followed the mechanism suggested by [17], which uses an Endpoint objects to the network traffic information of nodes.

The base of the THPA algorithm is described by Algorithm 2. Similar to KHPA, THPA periodically collects the resource metric to determine whether scaling is necessary (lines 2–6 in Algorithm 2). Once the scaling is decided, THPA calculates the number of pods to be adjusted according to the traffic distribution. For upscaling, THPA distributes new pods to worker nodes proportionally to the current traffic information (lines 7–14 in Algorithm 2), and that information is stored in *podsToAddAtNode*, which will then be referred to by the scheduler in Algorithm 3. For example, in Fig. 1 (b), assuming that the desired number of pods (*dPods*) in line 6 in Algorithm 2 is 6, THPA calculates the distribution of three new pods according to the 7:1:1 proportion of traffic. By dividing the *totalTraffic*, the distribution of new pods (*podsToAddAtNodes*) becomes 3-0-0, which indicates that worker 1 is assigned three more pods to handle concentrated traffic, while workers 2 and 3 keep the same numbers of pods.

Algorithm 2 *Calculating Number of Pods Will Be Created/Terminated at Each Worker Nodes*

pods : *list of application pods*.
curMetricVal : *current metric value*.
dMetricVal : *desired metric value*.
curPods : *current number of application pods*.
dPods : *desired number of application pods*.
HPA_Sync_Period : *HPA period*.
nodes : *list of worker nodes*.
nodesTraffic : *contains traffic information of worker nodes*.

```
1: while True do
2:     curPods = getCurPods(app)
3:     curMetricVal = getCurMetricValue(app)
4:     dMetricVal = getDesiredMetricValue(app)
5:     ratio = curMetricVal ÷ dMetricVal
6:     dPods = ceil(ratio × curPods)
7:     if dPods > curPods then                          ▷ Upscale
8:         nPodsToAdd = dPods − curPods
9:         nodesTraffic = getCurTraffic()
10:        totalTraffic = sum(nodesTraffic)
11:        for node ∈ nodes do
12:            prop = nodesTraffic[node]/totalTraffic
13:            podsToAddAtNode[node] +=
                                        prop ∗ nPodsToAdd
14:        end for
15:    else if dPods < curPods then                     ▷ Downscale
16:        nPodsToTerm = curPods − dPods
17:        nodesTraffic = getInverseCurTraffic()
18:        totalTraffic = sum(nodesTraffic)
19:        for node ∈ nodes do
20:            prop = nodesTraffic[node]/totalTraffic
21:            podsToTermAtNode[node] +=
                                      prop ∗ nPodsToTerm
22:        end for
23:        allPods = getAllPods(app)
24:        setTerminateAnno(allPods, False)
25:        for pod ∈ allPods do
26:            node = pod.getNode()
27:            if podsToTermAtNode[node] > 0 then
28:                setTerminateAnno(pod, True)
29:                podsToTermAtNode[node] −= 1
30:            end if
31:        end for
32:    end if
33:    setDesiredPods(app, dPods)
34:    time.sleep(HPA_Sync_Period)
35: end while
```

Let n denote the number of nodes in the cluster. The upscaling process (lines 7–14 in Algorithm 2) assigns new pods to each individual node proportionally to the traffic, and this process is iterated for all n node. Thus, the time complexity of the upscaling process in Algorithm 2 is O(n).

Algorithm 3 *Selecting Node for Scheduling Pod*

Required :
Desired application pods > current application pods.

```
1: unScheduledPod = getNextPodToSchedule(app)
2: if unScheduledPods! = null then
3:     filtering()
4:     scoring()
5:     /∗∗Extender logic
6:     for node ∈ nodes do
7:         if podsToAddAtNode[node] > 0 then
8:             assignPodToNode(pod, node)
9:             podsToAddAtNode[node] −= 1
10:            return
11:        end if
12:    end for
13:    ∗∗/
14: end if
```

Algorithm 4 *Choosing Pods for Termination*

Required :
Desired application pods < current application pods.
Termination annotation is set for all application pods.

```
1: pods = getAllPods(app)
2: podsToTerminate = []
3: for pod ∈ pods do
4:     if pod.getTerminateAnno() == True then
5:         podsToTerminate.add(pod)
6:     end if
7: end for
8: for pod ∈ podsToTerminate do
9:     terminate(pod)
10: end for
```

Algorithm 3 describes how the scheduler schedules new pods to the worker nodes in a cluster during the upscaling process. It is important to note that the scheduler finds the best node to place a pod according to scheduling policies, such as predicates and priorities, through filtering and scoring steps (lines 3–4 in Algorithm 3). In THPA, to overwrite these policies and schedule new pods according to *podsToAddAtNode* decided by THPA, the scheduler extender, described in lines 6–12 in Algorithm 3 is used. Regarding the time complexities, it is clear that those of the filtering and scoring steps in the worst case with n nodes are O(n) because they must iterate over all n workers in a cluster to filter and score them. For the extender logic, the algorithm traverses over all workers to find the intended node for scheduling new pods, so the time complexity for the extender logic is also O(n). Therefore, the time complexity of Algorithm 3 is O(n).

The downscaling process also requires information about network traffic accessing worker nodes. However, we inverse the value of traffic to make nodes with lower incoming traffic

have more pods to terminate (line 17 in Algorithm 2). Then, THPA calculates the number of pods to terminate for each node based on the reversed traffic (lines 19–22 in Algorithm 2). Note that we use the termination annotation to notify the Replicaset controller that is responsible for terminating pods in Algorithm 4. Thus, all pods are initialized as False, as in line 24 in Algorithm 2, and pods stored in *podsToTermAtNode* are set to True (lines 25–31 in Algorithm 2) so that the Replicaset controller can refer to these in Algorithm 4. Let n and p denote the numbers of nodes and pods in the cluster, respectively. The downscaling process iterates n times and then p times to calculate the number of pods to terminate at each node (lines 19–22 in Algorithm 2) and to set the termination annotation for an individual pod (lines 25–31 in Algorithm 2). Therefore, the time complexity of the downscaling process is O(n+p).

For example, assume that the cluster has three worker nodes and three pods are running on each node. If the current traffic is reduced to 1:1:7 and the number of pods to terminate (*nPodsToTerm*) in line 16 in Algorithm 2 is given as 4, THPA calculates the number of pods to terminate at each node based on the inversed traffic (1:1:1/7), and the resulting number of pods to terminate is 2-2-0 (*podsToTermAtNode*). Then, THPA refers to *podsToTermAtNode* to set the termination annotation values of two pods at worker 1 and two pods at worker 2 to True. Finally, the Replicaset controller terminates all pods with termination value set to True.

Note that the Replicaset controller is designed by default to select the pods in the cluster to terminate according to criteria in the order of assignment status of pod, phase of pod, readiness of pod, pod's ready transition time, pod's container restart count, and pod's creation time. Because it is impossible to designate a worker node of a pod to terminate with default criteria, so we newly define the termination annotation [36] in Algorithms 2 and 4 and put this annotation on top of the comparison criteria in pod termination selection in the Replicaset controller. Therefore, THPA can control the pods set to terminate, which will be chosen by the Replicaset controller, and we can conclude that THPA adjusts the number of pods at worker nodes in both downscaling and upscaling. Algorithm 4 iterates over all targeted application pods and then checks for their termination annotation to select which pods will be terminated. Thus, with p pods in the cluster, the time complexity of Algorithm 4 is O(p).

V. PERFORMANCE EVALUATION

In this section, we evaluate the performance of THPA in terms of throughput and response time in diverse scenarios. A Kubernetes cluster with three worker nodes was set up as shown in Fig. 2 with versions of Kubernetes and Docker as 1.18.0 and 19.03.13, respectively. The master node was configured with 4 CPU cores and 8 GB of RAM, while worker nodes were configured with 4 CPU cores and 4 GB of RAM. Another machine called a traffic generator was set up with 6 CPU cores and 8 GB of RAM, and the Apache HTTP

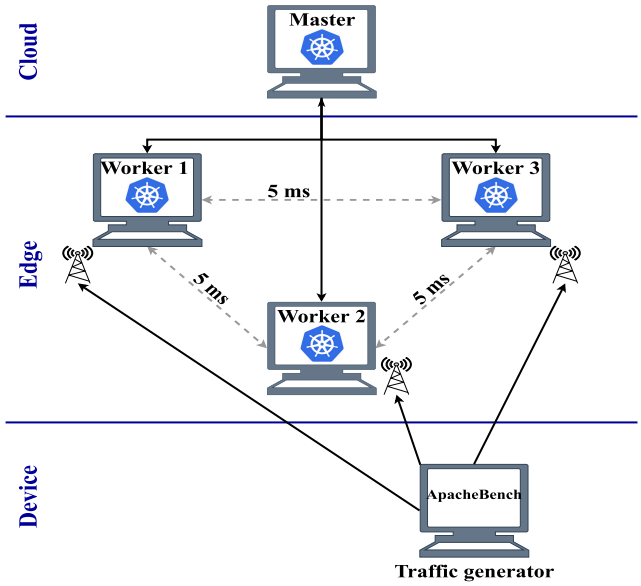

FIGURE 2. Kubernetes-based Edge computing testbed.

server benchmarking tool [37] was installed to generate requests to the application. To simulate the geographical distribution of edge nodes in the edge computing environment and to see the effect of delay clearly, the round-trip delay time between worker nodes was set to 10 ms. A simple web application was deployed on worker nodes, and it was exposed to the clients via the NodePort service. Furthermore, the minimum number of pods, maximum number of pods, and target CPU utilization values of KHPA and THPA were set as 3, 12, and 80%, respectively.

A. POD AUTOSCALING IN REAL-TIME

Fig. 3 illustrates the pod adjustment processes of KHPA and THPA according to the fluctuation of network traffic accessing worker nodes over time. Overall, THPA adjusts the number of application pods in a cluster according to the actual network traffic at worker nodes, whereas KHPA does not. For the first 30 s (from the 0th to 30th second), as the proportion of concurrent requests accessing worker nodes was 1:1:1, and the pods distribution in cluster remained at 1-1-1 for both KHPA and THPA. This is because the current number of pods in a cluster was sufficient to handle the incoming network traffic, so neither KHPA nor THPA performed the scaling. From 30–150 s, as the number of concurrent requests accessing worker nodes increased from 1:1:1 to 8:8:8, both KHPA and THPA allocated three more pods to each worker node so that the distribution of pods in the cluster became 4-4-4. At this stage, both KHPA and THPA evenly allocated new pods to worker nodes to handle the increased traffic. From 150–570 s, the numbers of concurrent requests accessing workers 2 and 3 fell to zero, whereas the incoming traffic at worker 1 remained the same. This caused both KHPA and THPA to remove five pods from the cluster, which led to the distributions of pods becoming 2-3-2 for KHPA and 4-2-1

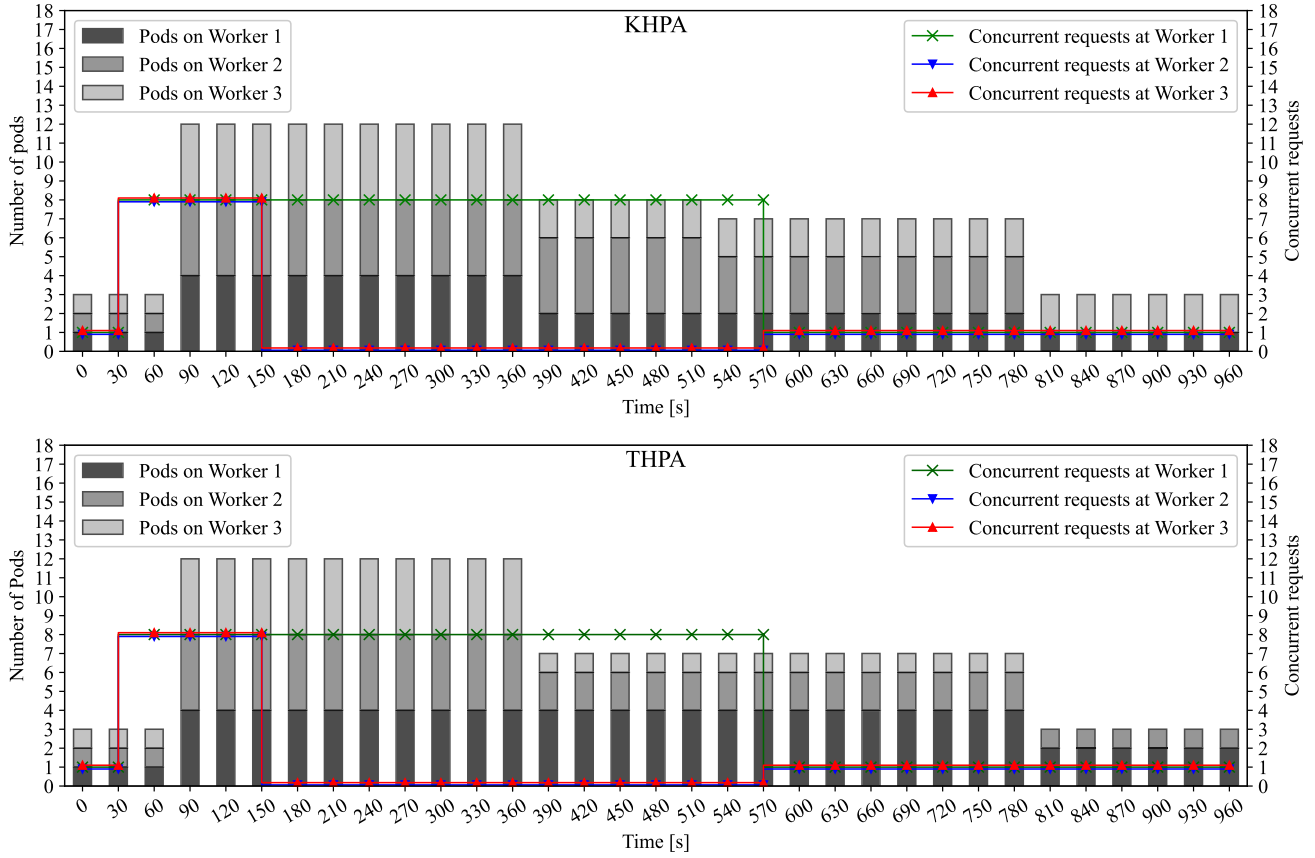

FIGURE 3. Pod scaling in real time.

for THPA (at 570 s). From the obtained pods distributions, KHPA continued to show its unawareness of traffic when it terminated two pods on worker 1 although high traffic was accessing it. Meanwhile, THPA maintained the number of pods at worker 1 and prioritized terminating five pods on workers 2 and 3 because there was no traffic accessing them.

For the remaining time (from 570–960 s), KHPA and THPA continued to scale down the application pods from seven at 570 s to three at 810 s according to the continuous fall of network traffic from 8:0:0 to 1:1:1. After downscaling, the distribution of pods of KHPA became 1-2-0, which emphasized that KHPA only relies on pod status to perform the downscaling. Meanwhile, the distribution of pods of THPA became 2-1-0. This is because THPA considered that the traffic accessing nodes was the same, so it terminated two pods on worker 1 and one pod for each of workers 2 and 3 because the number of pods to terminate was 4, which can not be fully divided by three, and the number of remaining pods on worker 3 before the downscaling was 1.

B. POD SCALING ACCORDING TO NETWORK TRAFFIC ACCESSING CLUSTER

Table 1 summarizes how KHPA and THPA allocated new pods to worker nodes according to different network traffic scenarios. In scenario 1, when there was one concurrent request accessing each worker node, both KHPA and THPA did not adjust the number of application pods in the cluster because the current traffic did not make the average CPU usage of pods exceeds the scaling threshold. However, except scenario 1, both KHPA and THPA increased the number of application pods in the cluster to adapt to the increase in network traffic accessing worker nodes. For example, as shown in Table 1, the total number of pods in the cluster increased from 6 in scenarios 2–4 to 12 in scenarios 8–10 to handle heavy traffic accessing worker nodes. It is remarkable that KHPA evenly distributed new pods to worker nodes regardless of the network traffic information, while THPA distributed new pods to worker nodes according to the traffic volume accessing them. For example, for scenarios 2–4 requiring three more pods in the cluster, KHPA only evenly distributed new pods to the three worker nodes regardless of the change in proportion of traffic. On the contrary, THPA allocated three additional pods proportionally to the traffic ratio accessing worker nodes. This is why the pod distribution in scenario 3 became 3-2-1, while that in scenario 4 was 4-1-1. It is interesting to note that THPA considers the deployment of new pods and termination of existing pods, not all the application pods in the cluster, to provide a seamless service.

TABLE 1. Pod autoscaling according to the proportion of network traffic.

Scenario	Proportion of concurrent requests [w1:w2:w3]	Total number of pods in cluster	Distribution of pods after scaling	
			KHPA [w1-w2-w3]	THPA [w1-w2-w3]
1	1:1:1	3	1-1-1	1-1-1
2	3:3:3	6	2-2-2	2-2-2
3	6:2:1	6	2-2-2	3-2-1
4	7:1:1	6	2-2-2	4-1-1
5	5:5:5	9	3-3-3	3-3-3
6	10:4:1	9	3-3-3	5-3-1
7	13:1:1	9	3-3-3	7-1-1
8	8:8:8	12	4-4-4	4-4-4
9	16:7:1	12	4-4-4	7-4-1
10	22:1:1	12	4-4-4	10-1-1

TABLE 2. Comparison between KHPA and THPA.

Comparison factor	KHPA	THPA
Supported metrics	CPU, Memory and custom metrics.	
Traffic-aware	No	Yes
Upscale	Evenly distributes new pods to nodes regardless of network traffic information.	Distributes new pods to nodes according to network traffic distribution.
Downscale	Terminates pods on nodes based on pods' status, such as the assignment status, phase, etc without considering network traffic information.	Terminates pods on nodes according to network traffic distribution.

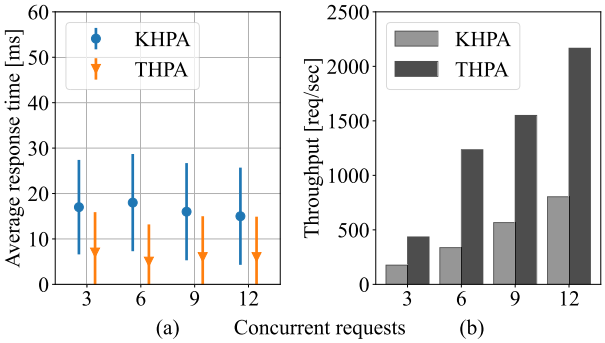

FIGURE 4. Application performance at worker 1 (a) Response time, and (b) Throughput.

Therefore, the pod distribution might not exactly match to the proportion of traffic accessing worker nodes. Table 2 summarizes the differences between KHPA and THPA based on the achieved results presented in Sections V-A and V-B. Furthermore, the effect of pod distribution is discussed in the next subsections.

C. EFFECT OF PODS DISTRIBUTION ON APPLICATION PERFORMANCE

To demonstrate that the application performance can be substantially affected by the pod distribution in an edge computing environment, we evaluated the throughput and average response time of the application while varying the traffic distribution.

1) EFFECT OF POD DISTRIBUTION ON APPLICATION PERFORMANCE AT ONE WORKER NODE

Fig. 4 (a) and (b) compare the average response times and throughputs of an application at worker 1 by increasing the number of concurrent requests accessing worker 1. Note that the pods distributions of KHPA and THPA were fixed at 3-3-3 and 7-1-1, respectively, to focus on the effect of pod distribution.

Overall, in Fig. 4 (a), the average response time of KHPA slightly fluctuated from 15 to 19 ms regardless of the number of concurrent requests accessing worker 1 increasing from 3 to 12. Meanwhile, the average response time of THPA only varied from 5 to 7.5 ms, which is approximately a 150% improvement over KHPA.

It is important to remember that the kube-proxy evenly distributes incoming traffic to all pods in the cluster. Therefore, although the incoming requests concentrated on worker 1 in this evaluation, they were handled by all pods in the cluster, and the throughput could differ according to the distribution of pods. More specifically, the number of application pods on worker 1 for KHPA was 3, which indicates that the number of incoming requests that could be locally handled at worker 1 for KHPA was only 3/9, whereas that of THPA was 7/9 because the number of pods on worker 1 for THPA was 7. In other words, 6/9 of the incoming requests at worker 1 for KHPA were redirected to other pods on remote nodes by the kube-proxy, while that of THPA was oly 2/9. From these facts and the existence of network delay between nodes, it is straightforward that a large proportion of incoming requests at worker 1 for KHPA required more time for responding, which directly increased the application response time at worker 1. In contrast, the higher number of pods at worker 1 for THPA effectively reduced the number of redirected requests at worker 1, such that the application response time at worker 1 could be improved by approximately 150% compared to KHPA. Consequently, in Fig. 4 (b), although the throughput of application of KHPA and THPA tended to increase as the number of concurrent requests accessing worker 1 increased, the obtained throughput at worker 1 of THPA was approximately 150% higher than at KHPA in all concurrent request cases. This is because the number of incoming requests that could be locally handled in THPA was higher than that by KHPA, as we already

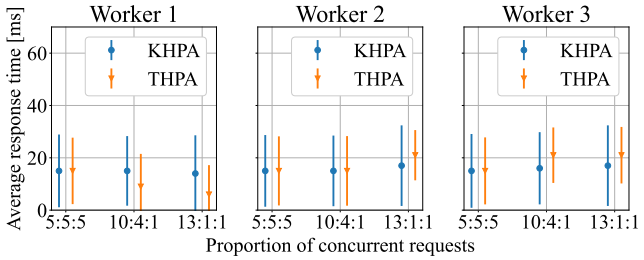

FIGURE 5. Application response time at three worker nodes.

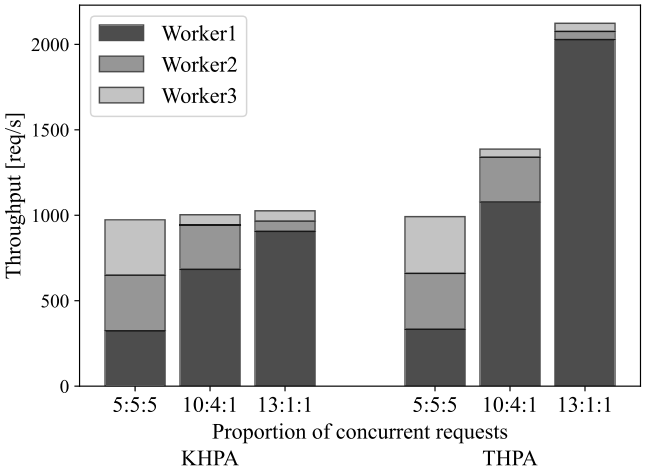

FIGURE 6. Application aggregated throughput at three worker nodes.

discussed for Fig. 4 (a). In other words, THPA reduced the network latency effect on application throughput at worker 1 by allocating more pods to worker 1 as the traffic accessing it increased. This result proves that application performance at specific worker node can be significantly affected by the distribution of pods in the cluster.

2) EFFECT OF POD DISTRIBUTION ON APPLICATION PERFORMANCE AT CLUSTER SCALE

Fig. 5 and 6 compare the response time and aggregated application throughput in scenarios 5, 6, and 7 from Table 1, where the pod distributions of THPA for the 5:5:5, 10:4:1, and 13:1:1 network traffic proportions were 3-3-3, 5-3-1, and 7-1-1, respectively, while those of KHPA were 3-3-3 for all traffic cases.

Fig. 5 describes the average response time at each worker according to three traffic distributions. When the traffic was balanced, such as 5:5:5, the pod distributions of both KHPA and THPA were 3-3-3, and the response time was approximately 14.4 ms irrespective of worker nodes. However, as the network traffic became imbalanced, such as 10:4:1 or 13:1:1, the difference between the average response times of KHPA and THPA increased. For example, the average response time at worker 1 for KHPA was approximately 60% higher than that for THPA in the 10:4:1 case. Moreover, that difference increased to approximately 130% in the 13:1:1 case. This is because the number of pods on worker 1 for THPA was 7, which is higher than that for KHPA, which was 3. This indicates that a higher portion of incoming requests, 7/9 at worker 1, were locally handled in THPA compared to KHPA, where only 3/9 were locally handled. In other words, THPA can reduce the average response time by approximately 38% by allocating more new pods to worker 1. Nevertheless, at workers 2 and worker 3, the average application response times of THPA were slightly higher than those of KHPA because the numbers of application pods at workers 2 and 3 were lower than KHPA.

Fig. 6 illustrates the aggregate throughputs of KHPA and THPA according to traffic distribution. From the figure, we can observe that with the balanced traffic distribution 5:5:5, the aggregated throughputs of KHPA and THPA were the same. Meanwhile, when the traffic distribution became imbalanced, such as 10:4:1 and 13:1:1, the aggregated throughput of THPA was approximately 38% and 100% higher than that of KHPA, respectively.

It is important to note that the distribution of pods of KHPA was always 3-3-3 regardless of the change of traffic distribution. Because the kube-proxy distributes incoming requests evenly to worker nodes based on a round-robin algorithm, the proportion of incoming requests at worker 1 that can be locally handled in KHPA is always 3/9. In other words, 6/9 of the incoming requests at worker 1 for KHPA are redirected to remote nodes for processing and such redirection significantly worsens the effect of network delay between nodes in application response time at worker 1. Hence, even when the incoming requests at worker 1 increase, the obtained throughput at worker 1 of KHPA can not increase significantly. Meanwhile, because THPA allocates new application pods to the worker nodes according to the traffic distribution, the number of new pods distributed to worker 1 also increases as the traffic accessing worker 1 increases. Therefore, the proportion of incoming requests that can be locally handled by THPA is maximized for each traffic distribution, and the response time at worker 1 can be reduced. For instance, when the traffic distribution was 13:1:1, the aggregated throughput of THPA was approximately 100% higher than that of KHPA. This is because the number of pods at worker 1 in THPA was 7, which is higher than that in KHPA, which was 3. Hence, the proportion of incoming requests that could be locally handled at worker 1 for THPA was 7/9, whereas that for KHPA was only 3/9. In other words, the incoming requests at worker 1 for KHPA required more time for response because 6/9 of them were redirected to remote nodes for processing, while that for THPA was only 2/9. Therefore, the obtained throughput at worker 1 for THPA was approximately 125% higher than that for KHPA because the number of redirected requests was minimized. It is important to note that because there was only one request accessing worker 2 and 3, although the number of pods on workers 2 and 3 for KHPA was three time higher than for THPA (3 compared to 1), the obtained throughput at workers 2 and 3 for KHPA was just slightly higher than that for THPA. Therefore, we can conclude that adjusting the number of application pods on worker nodes based on the network traffic accessing them can significantly improve

the response time and throughput of application in cluster by maximizing the number of incoming requests that can be locally handled and minimizing the effect of network delay between nodes in an edge computing environment.

VI. CONCLUSION

Edge computing infrastructure has emerged to address the challenge of handling a massive number of IoT devices in many IoT applications requiring low response time. Kubernetes provides flexible and powerful features to support IoT services in edge computing; in particular, the KHPA provides dynamic autoscaling for real-time service demand. In this paper, we showed that KHPA is not suitable for an edge computing environment where edge nodes are geographically dispersed and the amounts of traffic accessing nodes are imbalanced because new pods are scheduled to nodes regardless of traffic distribution. To overcome this problem, we proposed THPA, which can maximize the amount of traffic handled locally as well as minimize the long round-trip delay in an edge computing environment. The experimental results show that THPA dynamically adjusts the number of pods in the cluster according to the network traffic distribution accessing nodes in both upscaling and downscaling, resulting in significant IoT services quality improvement. More specifically, THPA not only provides better performance regardless of traffic distribution but also improves throughput and response time by approximately 150% compared to KHPA as the number of simultaneous requests increases. Therefore, we can conclude that it is important to provide the proper resource scaling according to the network traffic at each edge node to maximize IoT applications performance in an edge computing environment.

REFERENCES

[1] I. Martinez, A. Jarray, and A. S. Hafid, "Scalable design and dimensioning of fog-computing infrastructure to support latency-sensitive IoT applications," *IEEE Internet Things J.*, vol. 7, no. 6, pp. 5504–5520, Jun. 2020.

[2] A. A. Sadri, A. M. Rahmani, M. Saberikamarposhti, and M. Hosseinzadeh, "Fog data management: A vision, challenges, and future directions," *J. Netw. Comput. Appl.*, vol. 174, Jan. 2021, Art. no. 102882.

[3] M. Whaiduzzaman, M. J. N. Mahi, A. Barros, M. I. Khalil, C. Fidge, and R. Buyya, "BFIM: Performance measurement of a blockchain based hierarchical tree layered fog-IoT microservice architecture," *IEEE Access*, vol. 9, pp. 106655–106674, 2021.

[4] S. Aljanabi and A. Chalechale, "Improving IoT services using a hybrid fog-cloud offloading," *IEEE Access*, vol. 9, pp. 13775–13788, 2021.

[5] N. Al-Nabhan, S. Alenazi, S. Alquwaifili, S. Alzamzami, L. Altwayan, N. Alaloula, R. Alowaini, and A. B. M. A. A. Islam, "An intelligent IoT approach for analyzing and managing crowds," *IEEE Access*, vol. 9, pp. 104874–104886, 2021.

[6] H. Xu, W. Huang, Y. Zhou, D. Yang, M. Li, and Z. Han, "Edge computing resource allocation for unmanned aerial vehicle assisted mobile network with blockchain applications," *IEEE Trans. Wireless Commun.*, vol. 20, no. 5, pp. 3107–3121, May 2021.

[7] J. Hwang, L. Nkenyereye, N. Sung, J. Kim, and J. Song, "IoT service slicing and task offloading for edge computing," *IEEE Internet Things J.*, vol. 8, no. 14, pp. 11526–11547, Jul. 2021.

[8] M. Bukhsh, S. Abdullah, and I. S. Bajwa, "A decentralized edge computing latency-aware task management method with high availability for IoT applications," *IEEE Access*, vol. 9, pp. 138994–139008, 2021.

[9] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the Internet of Things," in *Proc. 1st, Ed., Workshop Mobile Cloud Comput. (MCC)*, 2012, pp. 13–16.

[10] A. Abouaomar, S. Cherkaoui, Z. Mlika, and A. Kobbane, "Resource provisioning in edge computing for latency-sensitive applications," *IEEE Internet Things J.*, vol. 8, no. 14, pp. 11088–11099, Jul. 2021.

[11] J. Zhang, X. Zhou, T. Ge, X. Wang, and T. Hwang, "Joint task scheduling and containerizing for efficient edge computing," *IEEE Trans. Parallel Distrib. Syst.*, vol. 32, no. 8, pp. 2086–2100, Aug. 2021.

[12] K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, "KEIDS: Kubernetes-based energy and interference driven scheduler for industrial IoT in edge-cloud ecosystem," *IEEE Internet Things J.*, vol. 7, no. 5, pp. 4228–4237, May 2020.

[13] N. Nguyen and T. Kim, "Toward highly scalable load balancing in kubernetes clusters," *IEEE Commun. Mag.*, vol. 58, no. 7, pp. 78–83, Jul. 2020.

[14] R. Muddinagiri, S. Ambavane, and S. Bayas, "Self-hosted kubernetes: Deploying Docker containers locally with minikube," in *Proc. Int. Conf. Innov. Trends Adv. Eng. Technol. (ICITAET)*, Dec. 2019, pp. 239–243.

[15] T.-T. Nguyen, Y.-J. Yeom, T. Kim, D.-H. Park, and S. Kim, "Horizontal pod autoscaling in kubernetes for elastic container orchestration," *Sensors*, vol. 20, no. 16, p. 4621, Aug. 2020.

[16] Z. Ma, S. Shao, S. Guo, Z. Wang, F. Qi, and A. Xiong, "Container migration mechanism for load balancing in edge network under power Internet of Things," *IEEE Access*, vol. 8, pp. 118405–118416, 2020.

[17] N. D. Nguyen, L.-A. Phan, D.-H. Park, S. Kim, and T. Kim, "ElasticFog: Elastic resource provisioning in container-based fog computing," *IEEE Access*, vol. 8, pp. 183879–183890, 2020.

[18] *What is Kubernetes?* Accessed: Nov. 29, 2021. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[19] S. Taherizadeh and V. Stankovski, "Dynamic multi-level auto-scaling rules for containerized applications," *Comput. J.*, vol. 62, no. 2, pp. 174–197, Feb. 2019.

[20] 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.

[21] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *Proc. IEEE Conf. Netw. Softwarization (NetSoft)*, Jun. 2019, pp. 351–359.

[22] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards delay-aware container-based service function chaining in fog computing," in *Proc. IEEE/IFIP Netw. Oper. Manage. Symp. (NOMS)*, Apr. 2020, pp. 1–9.

[23] Ł. Wojciechowski, K. Opasiak, J. Latusek, M. Wereski, V. Morales, T. Kim, and M. Hong, "NetMARKS: Network metrics-AwaRe kubernetes scheduler powered by service mesh," in *Proc. IEEE Conf. Comput. Commun. (INFOCOM)*, May 2021, pp. 1–9.

[24] D. Balla, C. Simon, and M. Maliosz, "Adaptive scaling of kubernetes pods," in *Proc. IEEE/IFIP Netw. Operations Manage. Symp. (NOMS)*, Apr. 2020, pp. 1–5.

[25] T. Hu and Y. Wang, "A kubernetes autoscaler based on pod replicas prediction," in *Proc. Asia–Pacific Conf. Commun. Technol. Comput. Sci. (ACCTCS)*, Jan. 2021, pp. 238–241.

[26] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for kubernetes edge clusters," *IEEE Trans. Netw. Service Manage.*, vol. 18, no. 1, pp. 958–972, Mar. 2021.

[27] J.-B. Lee, T.-H. Yoo, E.-H. Lee, B.-H. Hwang, S.-W. Ahn, and C.-H. Cho, "High-performance software load balancer for cloud-native architecture," *IEEE Access*, vol. 9, pp. 123704–123716, 2021.

[28] *Introduction to Kubernetes Architecture*. Accessed: Nov. 29, 2021. [Online]. Available: https://www.rancher.co.jp/learning-paths/introduction-to-kubernetes-architecture/

[29] A. Llorens-Carrodeguas, S. G. Sagkriotis, C. Cervelló-Pastor, and D. P. Pezaros, "An energy-friendly scheduler for edge computing systems," *Sensors*, vol. 21, no. 21, p. 7151, 2021.

[30] X. Zhang, L. Li, Y. Wang, E. Chen, and L. Shou, "Zeus: Improving resource efficiency via workload colocation for massive kubernetes clusters," *IEEE Access*, vol. 9, pp. 105192–105204, 2021.

[31] *Kubernetes Services*. Accessed: Nov. 29, 2021. [Online]. Available: https://www.vmware.com/topics/glossary/content/kubernetes-services

[32] (Jul. 2018). *IPVS-Based in-Cluster Load Balancing Deep Dive*. [Online]. Available: https://kubernetes.io/blog/2018/07/09/ipvs-based-in-cluster-load-balancing-deep-dive/

[33] *Scheduling Policies*. Accessed: Nov. 29, 2021. [Online]. Available: https://kubernetes.io/docs/reference/scheduling/policies/
[34] *Horizontal Pod Autoscaler | Kubernetes*. Accessed: Nov. 29, 2021. [Online]. Available: https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/
[35] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu, "Edge computing: Vision and challenges," *IEEE Internet Things J.*, vol. 3, pp. 637–646, May 2016.
[36] R. Treu. (Jan. 2020). *Kubernetes*. [Online]. Available: https://github.com/drbugfinder/kubernetes
[37] *Ab–Apache HTTP Server Benchmarking Tool-Apache HTTP Server Version 2.4*. Accessed: Nov. 29, 2021. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html

LE HOANG PHUC received the B.S. degree in electronics and communication engineering from Can Tho University, in 2018. He is currently pursuing the M.S. degree with the School of Information and Communication Engineering, Chungbuk National University, South Korea. His research interests include IoT applications, open-source software, and cloud/edge computing.

LINH-AN PHAN (Associate Member, IEEE) 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, South Korea, in 2019, where he is currently pursuing the Ph.D. degree with the School of Information and Communication Engineering. He worked as a Software Engineer at SVMC (Samsung Vietnam Mobile Research and Development Center), Vietnam, from 2013 to 2017. His research interests include IoT applications, open-source software, and cloud/edge computing.

TAEHONG KIM (Member, IEEE) received the B.S. degree in computer science from Ajou University, South Korea, in 2005, and the M.S. degree in information and communication engineering and the Ph.D. degree in computer science from the Korea Advanced Institute of Science and Technology (KAIST), in 2007 and 2012, respectively. He worked as a Research Staff Member at the Samsung Advanced Institute of Technology (SAIT) Samsung DMC Research and Development Center, from 2012 to 2014, and a Senior Researcher at the Electronics and Telecommunications Research Institute (ETRI), South Korea, from 2014 to 2016. Since 2016, he has been an Associate Professor with the School of Information and Communication Engineering, Chungbuk National University, South Korea. His research interests include edge computing, SDN/NFV, the Internet of Things, and wireless sensor networks. He has been an Associate Editor of IEEE Access, since 2020.