Delay-Aware Container Scheduling in Kubernetes

Wei-Kuang Lai, You-Chiun Wang, and Syu-Chen Wei

Abstract—Kubernetes is a powerful tool to manage containerized applications, which is also regarded as one promising platform to support microservices in edge computing. The scheduler is a key component of Kubernetes. It allocates each pod (i.e., a set of running containers) to one worker node (i.e., a machine). The default scheduler in Kubernetes is designed for the cloud environment containing homogeneous nodes. However, IoT edge nodes usually have various computing power and network bandwidth. The paper proposes a *delay-aware container scheduling (DACS) algorithm* to address the issue of node heterogeneity in edge computing. To efficiently assign pods to worker nodes, DACS takes account of not only residual resources of worker nodes but also potential delays caused by the pod assignment. We build a Kubernetes cluster by VMware to evaluate system performance. Experimental results reveal that DACS can significantly reduce both processing and network delays, thereby helping Kubernetes perform more efficiently in an edge environment.

Index Terms—container, delay, edge, Kubernetes, scheduling.

1 Introduction

An IoT (Internet of Things) system consists of many small computing devices that can self-organize and comprise a network. They keep sensing the surroundings and produce a massive amount of data to be sent through the network. Since it is not efficient to send all data to the cloud for processing, parts of the analytic work are transferred from the cloud to the edge [1]. However, distributing and managing loads to several hundred edge nodes is a big problem. A promising solution is to use containerized techniques such as Kubernetes.

Kubernetes, a framework developed for deploying and orchestrating containerized applications, is widely used in cloud-based web environments [2]. It also overcomes various difficulties that users may encounter when applying IoT solutions, like load management of IoT edge nodes. Kubernetes allocates resources to *containers*, which offer isolated contexts to carry out microservices. Containers should be carefully managed to distribute resources and loads, while ensuring both scalability and availability. To do so, Kubernetes uses the master-worker architecture and labels objects by name tags. It also performs jobs essential for reliably executing applications with efficient resource utilization, such as replication and failover.

Many studies [3]–[6] have pointed out the necessity and advantages of adopting Kubernetes in edge environments. More concretely, an edge environment is decentralized, which means that edge resources are more unstable, as compared to centralized cloud resources [7]. How to reliably manage edge resources and schedule tasks becomes a challenge. Kubernetes offers a highly available execution environment for containers and applies the Linux container technique [8] to support fault tolerance and auto-scaling. Hence, Kubernetes is well suited for managing unstable resources in an edge environment.

In Kubernetes, *pods* are considered as the smallest units of computing. A pod is a group of containers with shared storage and network resources, whose contents will be co-located and co-scheduled, and run in a shared context. Besides, a *worker node (WN)* is a machine used to host pods for execution, which can be either physical or virtual. Then, *scheduling* refers to how to assign pods to WNs. Kubernetes has a default scheduler (called *kube-scheduler*) that chooses a WN for each pod by a two-step method. The filtering step finds a set of WNs that have sufficient resources to satisfy the resource demand of a pod. Then, the scoring step ranks these WNs and assigns the pod to the highest-ranking WN.

The authors are with the Department of Computer Science and Engineering, National Sun Yat-sen University, Kaohsiung 80424, Taiwan (e-mail: wklai@cse.nsysu.edu.tw; ycwang@cse.nsysu.edu.tw; loerjeremywei@gmail.com).

TABLE 1: Comparison between KC scheduling and HD scheduling.

item	KC scheduling	HD scheduling
consideration	resources owned by WNs	capacities of instances
target	processing/network delays	task execution time
methodology	various (Section 3)	DAG or its variations

The kube-scheduler is competent for a cloud environment that contains homogeneous nodes with a similar number of resources. Nevertheless, IoT edge nodes are usually heterogeneous, which have diverse computing power, memory space, and network bandwidth. Hence, Kubernetes performance may degrade in an edge environment. To conquer the above problem, we propose a *delay-aware container scheduling (DACS) algorithm* in this paper. To assign pods to WNs more efficiently, DACS ponders not only residual resources of WNs but also potential delays due to the pod assignment. In particular, we consider two critical delays: the *processing delay* that each WN takes to handle the assigned pods, and the *network delay* caused by transmissions of container images and user data. To evaluate system performance, we create a Kubernetes cluster via the VMware Workstation [9]. Experimental results reveal that the DACS algorithm can keep both delays low, thereby reducing the amount of time required to complete all tasks, as compared with the existing solutions. Our contribution is to propose a novel scheduling algorithm to let Kubernetes work more efficiently in the edge environment by considering the heterogeneity of IoT edge nodes.

The Kubernetes container scheduling (called *KC scheduling*) discussed in this paper is different from the delay-sensitive scheduling in heterogeneous distributed environments (called *HD scheduling*). In HD scheduling, a set of distributed tasks with dependencies are dispatched to heterogeneous computing instances (e.g., CPUs or GPUs), whose primary objective is to minimize the execution time of tasks. In essence, there are three substantial differences between KC scheduling and

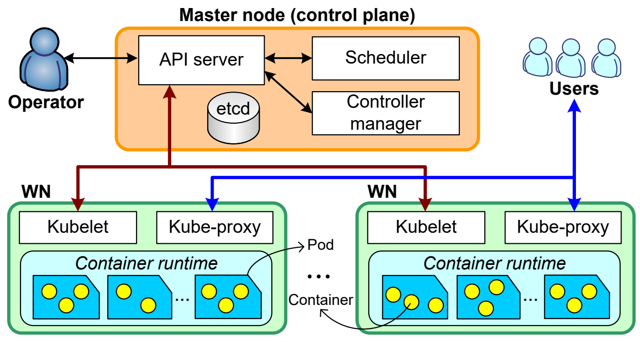


Fig. 1: Architecture of a Kubernetes cluster.

HD scheduling, as summarized in Table 1. First, KC scheduling considers the number of different resources (e.g., CPU, memory, and network) owned by WNs, while HD scheduling takes account of the capacities of computing instances. Second, KC scheduling aims to reduce both processing and network delays, whereas HD scheduling targets on saving task execution time. Third, there are various KC scheduling methods proposed, as discussed later in Section 3. On the other hand, most HD scheduling methods [10]–[12] are based on a *directed acyclic graph (DAG)* or its variations.

This paper is organized as follows: Section 2 introduces Kubernetes, Section 3 surveys the related work, and Section 4 gives the system model. We elaborate on the DACS algorithm in Section 5 and discuss some issues in Section 6. Afterward, the performance evaluation is presented in Section 7. Finally, Section 8 concludes this paper and gives future work.

2 Overview of Kubernetes

2.1 Kubernetes Architecture

Fig. 1 gives the architecture of a Kubernetes cluster, which contains one master node and multiple WNs. The master node (i.e., control plane) is responsible for allocating work to WNs and directing communications across the cluster. It has four major components. The *API server* offers internal and external interfaces to Kubernetes, which follows the representational state transfer (REST) style [13]. An operator employs the API server to cluster operations. The master node also uses it to communicate with WNs. The *scheduler* assigns pending pods to the WNs with enough resources to meet the users' requirements. The default scheduler is called the "kube-scheduler" (as discussed in Section 2.2), but it can be replaced by other plug-ins. The *controller manager* consists of multiple controllers. Each controller acts as a control loop that observes the shared state of the Kubernetes cluster via the API server and transfers the current state to the desired state by making some changes. Examples of controllers include the replication controller, endpoints controller, namespace controller, and service-accounts controller. The *etcd* is used to store data (e.g., configuration, state, and metadata). Any node in the cluster can read and write data through the etcd component.

WNs can host containers. A container is the lowest-level unit of the execution of an application, which holds running programs, libraries, and dependencies. There are three primary components in a WN. The *kubelet* ensures that the WN is in a running state and monitors the wholeness of every container. It takes care of the process of starting, stopping, and maintaining containers. The WN also interacts with the master node via its kubelet. The *kube-proxy* carries on the implementation of both load balancer and network proxy. It routes traffic to an adequate container based on the IP address and port number. Then, the *container runtime* creates and manages containers. To facilitate management, containers are grouped into pods.

2.2 Kubernetes Scheduler

When pods are created, they are added to a pending queue. Then, the scheduler iteratively fetches one pod from the queue and assigns it to a WN according to the pod's request. The default scheduler, namely kube-scheduler, employs a two-step method. In the *filtering step*, the kube-scheduler checks if the unused resources on a selected WN are sufficient. If not, the WN is filtered out. On the other hand, the *scoring step* ranks WNs that survived filtering by using the scoring equations:

$$\text{score} = \frac{S_{\text{CPU}} + S_{\text{MEM}}}{2} \times 10, \tag{1}$$

$$S_{\text{CPU}} = \frac{X_{\text{CPU}} - Y_{\text{CPU}}}{X_{\text{CPU}}} \text{ and } S_{\text{MEM}} = \frac{X_{\text{MEM}} - Y_{\text{MEM}}}{X_{\text{MEM}}}, \tag{2}$$

where X_{CPU} and X_{MEM} denote the WN's CPU and memory capacities, respectively. Besides, Y_{CPU} and Y_{MEM} are the number of CPU and memory resources requested by the WN's pods, respectively. After that, the kube-scheduler chooses the WN whose score is the highest to serve the pod. If there is a tie, a WN is randomly selected. In this way, the loads among WNs could be balanced in the long term [14].

The kube-scheduler is easy to implement and can be suitable for a cloud environment with homogeneous nodes. However, the computing power and network bandwidth of IoT edge nodes tend to be varied, which may result in different delays in processing and data transmissions. This issue is not addressed in the design of kube-scheduler.

2.3 Kubernetes Networking

Kubernetes networking takes charge of routing user requests between nodes in the cluster to their target pods, which relies on network plug-ins like Calico [15]. The iptables module [16] is used to control the network connections between pods. In Kubernetes, users can connect to their pods without keeping track of IP addresses. Furthermore, since each pod has its IP address and the containers within a pod listen to their native ports, the port mapping can be simplified or even ignored.

When a request reaches a WN, the WN's kube-proxy settles where the request is routed. If the target is a local pod on the same WN, the request is directly passed to the pod's interface. Otherwise, the request is forwarded to the assigned WN based on the L3 routing mechanism adopted and incorporated by the network plug-in. In this case, there will be a network delay. To evaluate this delay, we employ the average *round-trip time (RTT)* of packets between two nodes as a metric in our work.

3 Related Work

Different scheduling methods have been proposed for Kubernetes, which are divided into three categories. *Label-based scheduling methods* arrange pods through the label mechanism in Kubernetes. *Resource-based scheduling methods* handle pod allocation based on the number of resources owned by WNs. *Delay-based scheduling methods* take account of delays when assigning pods to WNs.

3.1 Label-based Scheduling Methods

Kubernetes offers a label mechanism to let pods be assigned to the WNs that meet certain conditions. For example, a user can request the scheduler to allocate the application to a WN with solid-state disks. Users need to know what types of labels are supported in a Kubernetes cluster.

In [17], four labels are employed for applications: high CPU usage, low CPU usage, high disk usage, and low disk usage. To reduce resource contention, the containers that wrap many applications with high CPU or disk usage will not be assigned to the same WN. The work [18] proposes a labeling system to customize labels, where a label indicates the usage degree (i.e., large, medium, and small) of a resource by each application. The scheduler assigns pods to WNs by referring to the labels of their applications. However, as labels hold only information on a rough amount of resource consumption by applications, the scheduler cannot assign pods to WNs very efficiently to improve their utilization.

3.2 Resource-based Scheduling Methods

Hu et al. [19] model the scheduling of containers through a vector bin packing problem. They assume that the containers of a distributed service have dependencies, and improve the tradeoff between load balance and dependency of WNs. The study [20] selects a WN for each container according to six criteria: CPU utilization, memory utilization, disk utilization, power consumption, the number of running containers, and the time required to send the container image. Two schedulers are proposed in [21]. The batch scheduler deals with tasks in rush time to increase throughput. The dynamic scheduler handles long-running tasks (e.g., deep-learning tasks) that are hungry for CPU resources, where their priorities may change.

With empirical profiling, Han et al. [22] analyze resource demands of the microservices for sensor data collection, WordPress, and online shopping, and use a greedy-based method to schedule containers based on the profiled result. The study [23] indicates that the overhead from network traffic and resource contention in disks may degrade Kubernetes performance, so it assigns containers that have dependencies in data to the same WN. Nguyen et al. [24] apply Kubernetes to fog computing, which allots resources to fog nodes according to the amount of network traffic that applications access at various locations. However, the above studies do not address the delay issue.

3.3 Delay-based Scheduling Methods

Some scheduling methods consider delays caused by different WN capabilities or network transmissions, which apply to edge environments. The work [25] assumes that more running containers on a WN postpone their completion time due to vying for the WN's resources. It estimates the contention rate (i.e., a measurement of degrees of resource competition) of each WN and then schedules containers to balance the resource contention on WNs. Fard et al. [26] aim to maximize system throughput, which is reflected by the number of completed microservices in a time interval, where the execution time of a microservice includes the scheduling latency, the waiting time for a WN, and the runtime of the microservice. However, both [25] and [26] consider only the processing delays on WNs.

Yin et al. [27] indicate that the completion time of a task will depend on the execution time on its WN and the network delay to transmit the task's container image and data. Given the deadlines of tasks, a scheduling approach is proposed to increase the number of accepted tasks (i.e., the tasks can be done within deadlines). Unlike the objective in [27], our work targets minimizing processing and network delays. The study [28] diminishes the service delivery time of each application, which is defined as the sum of data propagation and processing time for the application. The propagation time is estimated by the average input data size of the application divided by the network bandwidth. As compared with [28], we additionally consider the average RTTs between nodes, which offers a more accurate estimation of network delays. In [29], RTT values are assigned to WNs to be labels, which helps the scheduler place microservices in specific zones or suitable WNs based on the location delays. The study [30] applies the concept in [29] to deploying service function chains, where the containers of the same chain are placed on the adjacent WNs with lower RTT values. Both [29] and [30] designate some WNs as preferred nodes and assign pods to these WNs when they have enough resources. However, doing so would burden preferred nodes with heavy loads and thus increase their processing delays.

As compared with the above methods, our DACS algorithm assigns pods to WNs according to their resources and factors both processing and network delays incurred by assignments. DACS considers node heterogeneity, which is fairly common in edge computing. Taking node heterogeneity into consideration can greatly better container scheduling, which should not be ignored. Hence, DACS can make Kubernetes perform more efficiently in an edge environment.

4 System Model

We are given a Kubernetes cluster that contains a set $\hat{\mathcal{N}}$ of WNs. For each WN $n_i \in \hat{\mathcal{N}}$, a vector $\vec{R}_i = (r_i^{\mathrm{CPU}}, r_i^{\mathrm{MEM}}, r_i^{\mathrm{NET}})$ comprises its available resources, where r_i^{CPU}, r_i^{MEM}, and r_i^{NET} denote the number of n_i's residual CPU, memory, and network resources, respectively[1]. There is a set $\hat{\mathcal{P}}$ of pods in the cluster. For each pod $p_j \in \hat{\mathcal{P}}$, a vector $\vec{C}_j = (c_j^{\mathrm{CPU}}, c_j^{\mathrm{MEM}}, c_j^{\mathrm{NET}})$ presents its basic demand for resources, where c_j^{CPU}, c_j^{MEM}, and c_j^{NET} represent the minimum number of CPU, memory, and network resources that p_j needs to complete its task, respectively.

Suppose that a user u_k logins to a node $f_l(u_k)$, and u_k's application executes on a pod p_j assigned to a WN n_i, where $f_l(u_k) \in \hat{\mathcal{N}} \cup \{\tilde{m}\}$, $n_i \in \hat{\mathcal{N}}$, and $\tilde{m}$ is the master node. If $f_l(u_k) \neq n_i$, p_j is a remote pod; otherwise, p_j is a local pod. To efficiently assign pods to WNs, we should consider residual resources of WNs and also delays caused by pod assignments. Let $\tilde{D}_{\mathrm{p}}(u_k, p_j, n_i)$ and $\tilde{D}_{\mathrm{n}}(u_k, p_j, n_i)$ denote the *processing delay* and *network delay* of u_k's application running on pod p_j hosted by WN n_i (How to calculate these two delays will be discussed in Section 5.1). Besides, let $z_{k,j}^i$ be an indicator of whether u_k's application runs on pod p_j of WN n_i. If so, $z_{k,j}^i = 1$; otherwise, $z_{k,j}^i = 0$. Then, the scheduling problem can be expressed by

$$\text{Minimize } \frac{1}{|\hat{\mathcal{U}}|} \sum_{u_k \in \hat{\mathcal{U}}} \sum_{p_j \in \hat{\mathcal{P}}} \sum_{n_i \in \hat{\mathcal{N}}} z_{k,j}^i \times (\alpha \tilde{D}_{\mathrm{p}}(u_k, p_j, n_i) + (1-\alpha) \tilde{D}_{\mathrm{n}}(u_k, p_j, n_i)), \tag{3}$$

1. Similar to most studies discussed in Section 3, the CPU resource is the quantized computing power of the CPU, the memory resource is the space of the RAM, and the network resource is the network bandwidth.

TABLE 2: Summary of notations.

notation	definition
$\hat{\mathcal{U}}$	set of users (u_k: a user in $\hat{\mathcal{U}}$)
$\hat{\mathcal{P}}$	set of pods (p_j: a pod in $\hat{\mathcal{P}}$)
$\hat{\mathcal{N}}$	set of WNs (n_i: a WN in $\hat{\mathcal{N}}$)
$\hat{\mathcal{N}}_{\mathrm{CAN}}$, $\hat{\mathcal{N}}_{\mathrm{SEL}}$	candidate sets found by the scheduling and ranking modules ($\hat{\mathcal{N}}_{\mathrm{SEL}} \subseteq \hat{\mathcal{N}}_{\mathrm{CAN}} \subseteq \hat{\mathcal{N}}$)
$\vec{R}_i$	n_i's resource vector, $\vec{R}_i = (r_i^{\mathrm{CPU}}, r_i^{\mathrm{MEM}}, r_i^{\mathrm{NET}})$ for CPU, memory, and network resources
$\vec{C}_j$	p_j's demand vector, $\vec{C}_j = (c_j^{\mathrm{CPU}}, c_j^{\mathrm{MEM}}, c_j^{\mathrm{NET}})$ for CPU, memory, and network resources
α	coefficient to adjust the weights of delays
$z_{k,j}^i$	indicator to check if u_k's app runs on p_j of n_i
$\tilde{D}_{\mathrm{p}}(u_k, p_j, n_i)$	processing delay of u_k's app on p_j hosted by n_i
$\tilde{D}_{\mathrm{n}}(u_k, p_j, n_i)$	network delay of u_k's app on p_j hosted by n_i
$\tilde{A}(c_j^{\mathrm{CPU}}, l_j^{\mathrm{CPU}}, r_i^{\mathrm{CPU}})$	the number of n_i's CPU resources given to p_j (l_j^{CPU}: limit of CPU resources given to p_j)
$f_l(u_k)$	the WN to which u_k logins
$f_v(u_k)$	data volume of u_k's app
$\tau(u_a, u_b)$	average RTT between two nodes u_a and u_b
ζ_i	the number of pods served by n_i
$\Omega_{i,j}$	average delay for n_i to serve p_j

app: application

subject to

$$0 \leq \alpha \leq 1, \tag{4}$$

$$z_{k,j}^i \in \{0, 1\}, \tag{5}$$

$$c_j^{\mathrm{CPU}} \leq r_i^{\mathrm{CPU}}, c_j^{\mathrm{MEM}} \leq r_i^{\mathrm{MEM}}, c_j^{\mathrm{NET}} \leq r_i^{\mathrm{NET}}, \text{ if } z_{k,j}^i = 1, \tag{6}$$

$$\sum\nolimits_{n_i \in \hat{\mathcal{N}}} z_{k,j}^i \leq 1, \ \forall p_j \in \hat{\mathcal{P}}. \tag{7}$$

The objective function in Eq. (3) aims to minimize the average delay for users to access their applications, where $\hat{\mathcal{U}}$ is the set of users. To add flexibility, we employ a coefficient α to adjust the weights of processing and network delays. For constraints, Eq. (4) shows that α is between 0 and 1, and Eq. (5) points out that $z_{k,j}^i$ is an indicator whose value is either 0 or 1. Eq. (6) gives the resource constraints, where WN n_i must have enough CPU, memory, and network resources to meet pod p_j's basic requirement (if p_j is served by n_i). Then, Eq. (7) means that each pod can be placed on only a WN at most if designated. The above formulation is in the form of mixed-integer linear programming (MILP), so the scheduling problem is NP-hard. Table 2 summarizes the notations used in this paper.

5 The Proposed DACS Algorithm

Our DACS algorithm has two modules. For each pod p_j in $\hat{\mathcal{P}}$, the *scheduling module* filters out inappropriate WNs from $\hat{\mathcal{N}}$ according to p_j's resource demand (i.e., $\vec{C}_j$). Then, the *ranking module* grades the WNs that pass filtering brought up by the scheduling module, and presents it with the highest-rank WN to run p_j. To do so, the ranking module takes account of not only residual resources of WNs but also the processing and network delays induced by the pod assignment. Below, we discuss the calculation of delays, detail both scheduling and ranking modules, and give the innovation points of DACS.

5.1 Delay Calculation

Let $f_v(u_k)$ be the volume of data of a user u_k's application to be processed. When u_k's application runs on a pod $p_j \in \hat{\mathcal{P}}$ assigned to a WN $n_i \in \hat{\mathcal{N}}$, the processing delay is defined by

$$\tilde{D}_{\mathrm{p}}(u_k, p_j, n_i) = \frac{f_v(u_k)}{\tilde{A}(c_j^{\mathrm{CPU}}, l_j^{\mathrm{CPU}}, r_i^{\mathrm{CPU}})}, \tag{8}$$

where $\tilde{A}(c_j^{\mathrm{CPU}}, l_j^{\mathrm{CPU}}, r_i^{\mathrm{CPU}})$ is the number of CPU resources given to p_j. According to the Kubernetes documentation [31], each user can specify both demand (i.e., c_j^{CPU}) and limit (i.e., l_j^{CPU}) of CPU resources for his/her pod p_j, where $c_j^{\mathrm{CPU}} \leq l_j^{\mathrm{CPU}}$. If n_i has ample CPU resources, p_j can be allocated with more CPU resources to expedite its completion (but no more than l_j^{CPU} threshold). Hence, we adopt function $\tilde{A}(c_j^{\mathrm{CPU}}, l_j^{\mathrm{CPU}}, r_i^{\mathrm{CPU}})$ instead of demand c_j^{CPU} in the denominator of Eq. (8). For example, we can define the function as follows:

$$\tilde{A}(c_j^{\mathrm{CPU}}, l_j^{\mathrm{CPU}}, r_i^{\mathrm{CPU}}) = \max\{c_j^{\mathrm{CPU}}, \min\{\varphi r_i^{\mathrm{CPU}}, l_j^{\mathrm{CPU}}\}\}, \tag{9}$$

where $r_i^{\mathrm{CPU}} \geq c_j^{\mathrm{CPU}}$ and $0 < \varphi \leq 1$. In this way, we can avoid a pod consuming the most of n_i's residual CPU resources when l_j^{CPU} is set too large. Specifically, if $l_j^{\mathrm{CPU}} > \varphi r_i^{\mathrm{CPU}} > c_j^{\mathrm{CPU}}$, p_j is allocated with $\varphi r_i^{\mathrm{CPU}}$ CPU resources to speed up execution. On the other hand, when n_i's CPU resources are not plenteous (i.e., $c_j^{\mathrm{CPU}} > \min\{\varphi r_i^{\mathrm{CPU}}, l_j^{\mathrm{CPU}}\}$), p_j will be given the minimum guaranteed number c_j^{CPU} of CPU resources.

Depending on the location of pod p_j, there are two cases for calculating the network delay of u_k's application. If p_j is a local pod (i.e., $f_l(u_k) = n_i$), the network delay will be

$$\tilde{D}_{\mathrm{n}}(u_k, p_j, n_i) = t_j^{\mathrm{IM}} + f_v(u_k)/c_j^{\mathrm{NET}}. \tag{10}$$

The first term t_j^{IM} indicates the amount of time required to download p_j's container image. The second term $f_v(u_k)/c_j^{\mathrm{NET}}$ gives the amount of time taken to transmit u_k's application data. We use bandwidth demand c_j^{NET} in the denominator of the second term. On the other hand, when p_j is a remote pod (i.e, $f_l(u_k) \neq n_i$), we estimate the network delay as follows:

$$\tilde{D}_{\mathrm{n}}(u_k, p_j, n_i) = t_j^{\mathrm{IM}} + \frac{f_v(u_k)}{c_j^{\mathrm{NET}}} + \frac{\tau(n_i, f_l(u_k)) + \sigma_i}{2}. \tag{11}$$

In Eq. (11), the last term indicates the communication latency from n_i to $f_l(u_k)$, where $\tau(n_i, f_l(u_k))$ is the average RTT between n_i and $f_l(u_k)$, and σ_i is the standard deviation of RTTs between n_i and all other WNs in the Kubernetes cluster:

$$\sigma_i = \sqrt{\frac{\sum_{n_a \in \hat{\mathcal{N}} \setminus \{n_i\}} \tau(n_i, n_a)^2}{|\hat{\mathcal{N}}| - 1} - \mu_i^2}, \tag{12}$$

$$\mu_i = \frac{\sum_{n_b \in \hat{\mathcal{N}} \setminus \{n_i\}} \tau(n_i, n_b)}{|\hat{\mathcal{N}}| - 1}. \tag{13}$$

Specifically, considering that the network may be sometimes unstable, we thus add the standard deviation σ_i to the calculation of the last term in Eq. (11) to reflect network variation.

5.2 Scheduling Module

Algorithm 1 gives the pseudocode of the scheduling module. We use a variable ζ_i to record the number of pods that each WN $n_i \in \hat{\mathcal{N}}$ has served (this information is required by the ranking module, as discussed later in Section 5.3). The code in lines 1–2 initializes the ζ_i value. Then, the for-loop in lines 3–15 iteratively picks a pod p_j from $\hat{\mathcal{P}}$ and finds a suitable WN to cope with p_j, until all pods in $\hat{\mathcal{P}}$ have been checked. To do so, we maintain a candidate set $\hat{\mathcal{N}}_{\mathrm{CAN}}$ for p_j, which is initially set to $\hat{\mathcal{N}}$ (i.e., all WNs in the Kubernetes cluster), as shown

Algorithm 1: The Scheduling Module

```
Data: set $\hat{\mathcal{U}}$ of users, set $\hat{\mathcal{P}}$ of pods, and set $\hat{\mathcal{N}}$ of WNs
      in a Kubernetes cluster
Result: assignment of a WN in $\hat{\mathcal{N}}$ for each pod in $\hat{\mathcal{P}}$
1  foreach $n_i \in \hat{\mathcal{N}}$ do
2      $\zeta_i \leftarrow 0$;
3  foreach $p_j \in \hat{\mathcal{P}}$ do
4      $\hat{\mathcal{N}}_{\text{CAN}} \leftarrow \hat{\mathcal{N}}$;
5      $u_k \leftarrow$ the user in $\hat{\mathcal{U}}$ whose application runs on $p_j$;
6      foreach $n_i \in \hat{\mathcal{N}}_{\text{CAN}}$ do
7          if $\vec{R}_i <_{\mathbf{F}} \vec{C}_j$ then
8              $\hat{\mathcal{N}}_{\text{CAN}} \leftarrow \hat{\mathcal{N}}_{\text{CAN}} \setminus \{n_i\}$;
9      if $\hat{\mathcal{N}}_{\text{CAN}} = \emptyset$ then
10         $p_j$.state $\leftarrow$ pending;
11         continue;
12     $n_x \leftarrow \text{Ranking}(u_k, p_j, \hat{\mathcal{N}}_{\text{CAN}})$;
13     Assign pod $p_j$ to WN $n_x$;
14     $\zeta_x \leftarrow \zeta_x + 1$;
15     Update $\vec{R}_x$ based on $\vec{C}_j$;
```

in line 4. Moreover, let $u_k \in \hat{\mathcal{U}}$ be the user whose application runs on p_j. Like the filtering step in kube-scheduler, the code in lines 6–8 checks each WN n_i in $\hat{\mathcal{N}}_{\text{CAN}}$, and removes those WNs without sufficient CPU, memory, or network resources (i.e., $r_i^{\text{CPU}} < c_j^{\text{CPU}}$, $r_i^{\text{MEM}} < c_j^{\text{MEM}}$, or $r_i^{\text{NET}} < c_j^{\text{NET}}$) from $\hat{\mathcal{N}}_{\text{CAN}}$. Here, we denote by "$\vec{R}_i <_{\mathbf{F}} \vec{C}_j$" the above check in line 7.

However, if $\hat{\mathcal{N}}_{\text{CAN}}$ becomes empty after the filtering step in lines 6–8, which implies that no WN in the Kubernetes cluster has enough resources to serve p_j, we set p_j's state to *pending*. In this case, p_j will be scheduled in the next round. The code is given in lines 9–11. Otherwise, we find an appropriate WN from $\hat{\mathcal{N}}_{\text{CAN}}$ by the ranking module. Suppose that the ranking module returns a WN n_x. Then, we assign p_j to n_x, increase ζ_x by one (since n_x will serve p_j), and update n_x's resource vector $\vec{R}_x$ according to p_j's demand vector $\vec{C}_j$ as follows:

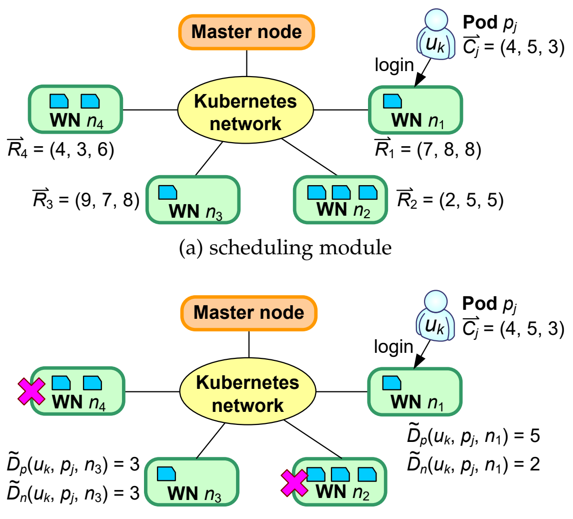
$$r_x^{\text{CPU}} = r_x^{\text{CPU}} - \tilde{A}(c_j^{\text{CPU}}, l_j^{\text{CPU}}, r_x^{\text{CPU}}), \tag{14}$$

$$r_x^{\text{MEM}} = r_x^{\text{MEM}} - \tilde{A}(c_j^{\text{MEM}}, l_j^{\text{MEM}}, r_x^{\text{MEM}}), \tag{15}$$

$$r_x^{\text{NET}} = r_x^{\text{NET}} - c_j^{\text{NET}}. \tag{16}$$

In Eqs. (14) and (15), n_x will allocate $\tilde{A}(c_j^{\text{CPU}}, l_j^{\text{CPU}}, r_x^{\text{CPU}})$ CPU resources and $\tilde{A}(c_j^{\text{MEM}}, l_j^{\text{MEM}}, r_x^{\text{MEM}})$ memory resources to p_j, as mentioned earlier in Section 5.1. Here, we can use Eq. (9) to define $\tilde{A}(c_j^{\text{MEM}}, l_j^{\text{MEM}}, r_x^{\text{MEM}})$ by replacing the term "CPU" with the term "MEM". Besides, n_x gives p_j the number of network resources that p_j requests (i.e., c_j^{NET}), as shown in Eq. (16).

Fig. 2(a) presents an example of a Kubernetes cluster with four WNs, where $\hat{\mathcal{N}} = \{n_1, n_2, n_3, n_4\}$. Suppose that a user u_k logins to WN n_1, whose application runs on pod p_j and p_j's demand vector is $\vec{C}_j = (4, 5, 3)$. Each WN has served some pods, and the resource vectors of WNs n_1, n_2, n_3, and n_4 are $\vec{R}_1 = (7, 8, 8)$, $\vec{R}_2 = (2, 5, 5)$, $\vec{R}_3 = (9, 7, 8)$, and $\vec{R}_4 = (4, 3, 6)$, respectively. According to Algorithm 1, the candidate set $\hat{\mathcal{N}}_{\text{CAN}}$ is initially set to $\{n_1, n_2, n_3, n_4\}$ by line 4. In lines 6–8, we filter out WNs without enough resources to serve p_j. Because $\vec{R}_2 <_{\mathbf{F}} \vec{C}_j$ and $\vec{R}_4 <_{\mathbf{F}} \vec{C}_j$, both WNs n_2 and n_4 are removed from $\hat{\mathcal{N}}_{\text{CAN}}$ (due to not enough CPU and memory resources, respectively). Hence, we obtain $\hat{\mathcal{N}}_{\text{CAN}} = \{n_1, n_3\}$, from which the ranking module in Section 5.3 will pick a WN to serve p_j. Then, Theorem 1 analyzes the time complexity of the scheduling module in Algorithm 1.



(a) scheduling module

(b) ranking module

Fig. 2: An example of the execution of the DACS algorithm.

Theorem 1. *Given ξ_P pods in $\hat{\mathcal{P}}$ and ξ_N WNs in $\hat{\mathcal{N}}$, Algorithm 1 takes $O(\xi_P(\xi_N + T_s))$ time in the worst case, where T_s is the time required to run the ranking module.*

Proof: In Algorithm 1, the first for-loop in lines 1–2 takes $O(\xi_N)$ time. All other statements are wrapped in the second for-loop (in lines 3–15), which repeats ξ_P times. Since $\hat{\mathcal{N}}_{\text{CAN}} \subseteq \hat{\mathcal{N}}$, we have $|\hat{\mathcal{N}}_{\text{CAN}}| \le |\hat{\mathcal{N}}| = \xi_N$. Hence, the inner for-loop in lines 6–8 spends no more than $O(\xi_N)$ time. Apart from line 12, which needs T_s time to run the ranking module, each residual statement takes a constant time. Thus, the time complexity is $O(\xi_N) + \xi_P(O(\xi_N) + T_s + O(1)) = O(\xi_P(\xi_N + T_s))$. □

5.3 Ranking Module

The ranking module picks the WN with the highest standing from candidate set $\hat{\mathcal{N}}_{\text{CAN}}$ to run pod p_j. Algorithm 2 gives the pseudocode. For each WN n_i in $\hat{\mathcal{N}}_{\text{CAN}}$, we calculate the delay $\tilde{D}_{\text{p}}(u_k, p_j, n_i)$ for n_i to process p_j by Eq. (8). If p_j is a local pod, we compute its network delay by Eq. (10); otherwise, we adopt Eq. (11). The code is given in lines 2–6. By combining processing and network delays, line 7 gives the average delay $\Omega_{i,j}$ for n_i to serve p_j, which corresponds to the objective function in Eq. (3). However, when n_i has been assigned with other pods (i.e., $\zeta_i > 0$), we have to consider the extra delay caused by *context switch* and *resource competition*, as denoted by Γ_i. How to estimate Γ_i will be discussed in Section 6.1.

Afterward, we rank each WN in $\hat{\mathcal{N}}_{\text{CAN}}$ based on its average delay $\Omega_{i,j}$, where the lower the average delay is, the higher the rank will be. Then, line 10 picks the WN with the highest rank, which is denoted by $n_{\texttt{ref}}$. Moreover, we also choose a subset $\hat{\mathcal{N}}_{\text{SEL}}$ of WNs from $\hat{\mathcal{N}}_{\text{CAN}}$ to be *final candidates*. Naturally, $n_{\texttt{ref}}$ should be the default member in $\hat{\mathcal{N}}_{\text{SEL}}$, as indicated by line 11. When a WN $n_i \in \hat{\mathcal{N}}_{\text{CAN}}$ ($n_i \ne n_{\texttt{ref}}$) satisfies the following condition, it is added to $\hat{\mathcal{N}}_{\text{SEL}}$:

$$\Omega_{i,j} \le \Omega_{\texttt{ref}} + \lambda, \tag{17}$$

Algorithm 2: The Ranking Module

```
Data: user $u_k$, pod $p_j$, and candidate set $\hat{\mathcal{N}}_{\texttt{CAN}}$ of WNs
Result: the selected WN to serve $p_j$
1  foreach $n_i \in \hat{\mathcal{N}}_{\texttt{CAN}}$ do
2      Calculate $\tilde{D}_{\text{p}}(u_k, p_j, n_i)$ by Eq. (8);
3      if $f_l(u_k) = n_i$ then
4          Calculate $\tilde{D}_{\text{n}}(u_k, p_j, n_i)$ by Eq. (10);
5      else
6          Calculate $\tilde{D}_{\text{n}}(u_k, p_j, n_i)$ by Eq. (11);
7      $\Omega_{i,j} \leftarrow \alpha\tilde{D}_{\text{p}}(u_k, p_j, n_i) + (1-\alpha)\tilde{D}_{\text{n}}(u_k, p_j, n_i)$;
8      if $\zeta_i > 0$ then
9          $\Omega_{i,j} \leftarrow \Omega_{i,j} + \Gamma_i$;
10 $n_{\texttt{ref}} \leftarrow \arg\min_{n_i \in \hat{\mathcal{N}}_{\texttt{CAN}}} \Omega_{i,j}$;
11 $\hat{\mathcal{N}}_{\texttt{SEL}} \leftarrow \{n_{\texttt{ref}}\}$;
12 Let $\Omega_{\texttt{ref}}$ be the average delay of $n_{\texttt{ref}}$;
13 foreach $n_i \in \hat{\mathcal{N}}_{\texttt{CAN}} \setminus \{n_{\texttt{ref}}\}$ do
14     if $\Omega_{i,j} \leq \Omega_{\texttt{ref}} + \lambda$ then
15         $\hat{\mathcal{N}}_{\texttt{SEL}} \leftarrow \hat{\mathcal{N}}_{\texttt{SEL}} \cup \{n_i\}$;
16 if $|\hat{\mathcal{N}}_{\texttt{SEL}}| > 1$ then
17     return $\arg\max_{n_i \in \hat{\mathcal{N}}_{\texttt{SEL}}} \min\{r_i^{\texttt{CPU}}, r_i^{\texttt{MEM}}\}$;
18 return $n_{\texttt{ref}}$;
```

where $\Omega_{\texttt{ref}}$ denotes the average delay of $n_{\texttt{ref}}$ and λ is a small constant. The idea behind Eq. (17) is that if n_i's average delay is merely *slightly* larger than $n_{\texttt{ref}}$, picking $n_{\texttt{ref}}$ or n_i doesn't make much of a difference (in terms of the average delay). In this case, we should view n_i as a final candidate, and refer to other metrics to compare $n_{\texttt{ref}}$ with n_i. The corresponding code is presented in lines 13–15.

If there is only one final candidate in $\hat{\mathcal{N}}_{\texttt{SEL}}$ (which is the best WN), the ranking module directly returns $n_{\texttt{ref}}$ by line 18. Otherwise, among all candidates in $\hat{\mathcal{N}}_{\texttt{SEL}}$, we select the WN with the maximum combination of residual CPU and memory resources. The code is given in lines 16–17. Intuitively, one may suggest adopting the sum of residual CPU and memory resources (i.e., $r_i^{\texttt{CPU}} + r_i^{\texttt{MEM}}$) to represent the combination. However, doing so may encounter some extreme conditions. Let us consider an example with two WNs n_a and n_b in $\hat{\mathcal{N}}_{\texttt{SEL}}$, where $(r_a^{\texttt{CPU}}, r_a^{\texttt{MEM}}) = (1, 9)$ and $(r_b^{\texttt{CPU}}, r_b^{\texttt{MEM}}) = (4, 5)$. Besides, p_j requires 1 CPU resource and 2 memory resources. Since $r_a^{\texttt{CPU}} + r_a^{\texttt{MEM}} > r_b^{\texttt{CPU}} + r_b^{\texttt{MEM}}$, n_a is chosen. However, n_a's CPU resources will be used up, leaving many memory resources wasted. Hence, we adopt $\min\{r_i^{\texttt{CPU}}, r_i^{\texttt{MEM}}\}$ in line 17 to represent the combination. In this case, $\min\{r_a^{\texttt{CPU}}, r_a^{\texttt{MEM}}\} < \min\{r_b^{\texttt{CPU}}, r_b^{\texttt{MEM}}\}$, so n_b is chosen, which is a better solution.

Fig. 2(b) continues the example discussed in Section 5.2, where $\hat{\mathcal{N}}_{\texttt{CAN}} = \{n_1, n_3\}$. As WN n_3 has more CPU resources than WN n_1, n_1 has a higher processing delay than n_3, that is, $\tilde{D}_{\text{p}}(u_k, p_j, n_1) > \tilde{D}_{\text{p}}(u_k, p_j, n_3)$. Since user u_k logins to n_1, pod p_j is a local pod for n_1 and a remote pod for n_3. Hence, n_1 has a lower network delay than n_3, that is, $\tilde{D}_{\text{n}}(u_k, p_j, n_1) < \tilde{D}_{\text{n}}(u_k, p_j, n_3)$. Let $\alpha = 0.5$, $\Gamma_1 = \Gamma_3 = 0.1$ (i.e., overhead of context switch and resource competition), and $\lambda = 0.05$. The average delays for both WNs to serve p_j will be

$$n_1: \quad \Omega_{1,j} = 0.5 \times 5 + 0.5 \times 2 + 0.1 = 3.6,$$

$$n_3: \quad \Omega_{3,j} = 0.5 \times 3 + 0.5 \times 3 + 0.1 = 3.1.$$

Since $\Omega_{1,j} > \Omega_{3,j} + \lambda$ (i.e., the condition in line 14 is not met), we thus assign p_j to n_3. Theorem 2 analyzes the time complexity of the ranking module in Algorithm 2.

Theorem 2. *Given ξ_C WNs in $\hat{\mathcal{N}}_{\texttt{CAN}}$, Algorithm 2 requires $O(\xi_C \log_2 \xi_C)$ time in the worst case.*

Proof: In Algorithm 2, the first for-loop (i.e., lines 1–9) repeats ξ_C times. The calculation of Eq. (8) and Eq. (10) take $O(1)$ time. In Eq. (11), we need the standard deviation σ_i of RTTs between a WN n_i and all other WNs, as computed by Eq. (12). Fortunately, σ_i can be calculated in advance before running the ranking module. Hence, line 6 takes $O(1)$ time. Since the code in lines 7–9 spends a constant time, the first for-loop takes $O(\xi_C)$ time. Finding a WN with the minimum average delay in line 10 requires $O(\xi_C \log_2 \xi_C)$ time, as we have to sort all WNs in $\hat{\mathcal{N}}_{\texttt{CAN}}$. The second for-loop in lines 13–15 takes $O(\xi_C - 1)$ time. Since $\hat{\mathcal{N}}_{\texttt{SEL}} \subseteq \hat{\mathcal{N}}_{\texttt{CAN}}$, we have $|\hat{\mathcal{N}}_{\texttt{SEL}}| \leq |\hat{\mathcal{N}}_{\texttt{CAN}}| = \xi_C$. Thus, line 17 spends $O(\xi_C \log_2 \xi_C)$ time in the worst case (due to sorting). To sum up, the total time complexity of Algorithm 2 is $O(\xi_C) + O(\xi_C \log_2 \xi_C) + O(\xi_C - 1) + O(\xi_C \log_2 \xi_C) = O(\xi_C \log_2 \xi_C)$. □

5.4 Innovation Points

The kube-scheduler could be efficient in a cloud environment with homogeneous nodes, where they have similar CPU and memory resources. However, IoT edge nodes are usually heterogeneous, which may degrade Kubernetes performance. As compared with the kube-scheduler, our DACS algorithm has three innovation points to help improve Kubernetes performance in an edge environment:

- In addition to CPU and memory resources (i.e., $r_i^{\texttt{CPU}}$ and $r_i^{\texttt{MEM}}$), DACS considers network resources (i.e., $r_i^{\texttt{NET}}$) of WNs. In this way, DACS can efficiently assign each pod p_j to a WN according to its bandwidth demand $c_j^{\texttt{NET}}$ to reduce the application's response time. This is especially important in the edge environment.
- DACS reflects WN heterogeneity in edge computing via processing delay $\tilde{D}_{\text{p}}(u_k, p_j, n_i)$ in Eq. (8) and network delay $\tilde{D}_{\text{n}}(u_k, p_j, n_i)$ in Eqs. (10) and (11). In addition, a coefficient α is used to adjust the weights of processing and network delays in the calculation of average delay, which adds more flexibility.
- Instead of forthright selecting the WN with the minimum average delay to serve a pod p_j, DACS's ranking module finds a set $\hat{\mathcal{N}}_{\texttt{SEL}}$ of WNs that have shorter average delays. Then, it picks the WN n_i from $\hat{\mathcal{N}}_{\texttt{SEL}}$ with the maximum combination of residual CPU and memory resources. Doing so has three benefits. First, a user (whose application running on a pod) will merely be served by a WN with enough CPU, memory, and network resources. In this way, a WN will exclusively utilize its resources on users that it can support. Second, since n_i has ample CPU or memory resources, n_i can allocate more CPU or memory resources to p_j than p_j requests to facilitate its execution. Third, we can reduce the *out-of-CPU* or *out-of-memory* probabilities. This situation may occur when some users underestimate or dynamically raise their CPU or memory usage (i.e., $c_j^{\texttt{CPU}}$ or $c_j^{\texttt{MEM}}$), as discussed in Section 6.2.

Theorem 3 analyzes DACS's time complexity.

Theorem 3. *Suppose that a Kubernetes cluster has ξ_N WNs and ξ_P pods. The worst-case time complexity of the DACS algorithm is $O(\xi_N(\xi_N + \xi_P \log_2 \xi_N))$.*

Proof: As mentioned in Theorem 2, the standard deviation σ_i of RTTs between a WN and other WNs can be calculated beforehand. According to Eqs. (12) and (13), it spends $O(\xi_N)$ time to find σ_i for a WN. Since there are ξ_N WNs, this part spends $O(\xi_N^2)$ time. By combining Theorems 1 and 2, the time complexity of DACS is $O(\xi_N^2) + O(\xi_P(\xi_N + \xi_C \log_2 \xi_C))$, where ξ_C is the number of WNs in $\hat{\mathcal{N}}_{\text{CAN}}$. As $\hat{\mathcal{N}}_{\text{CAN}}$ is a subset of $\hat{\mathcal{N}}$, we derive that $\xi_C \leq \xi_N$. Hence, the time complexity can be simplified to $O(\xi_N(\xi_N + \xi_P \log_2 \xi_N))$. □

6 Discussion

In this section, we discuss two issues. One is the resource competition when multiple pods are assigned to the same WN. The other is the dynamic environment where users may change resource usage at runtime.

6.1 Resource Competition

When multiple pods are scheduled to a WN, the resource competition (or resource contention) between these pods may occur if the total number of resources requested by them overtakes the WN's capacity. To handle the resource competition, Kubernetes allocates resources to pods on a first-come-first-served basis [32]. This causes extra delays, which rise rapidly as more pods are assigned to the WN and vie for resources.

In DACS, the scheduling module first filters out those WNs whose resources cannot satisfy a pod's demand (referring to lines 6–8 in Algorithm 1). By doing so, we can ensure that a WN has enough resources to support pods assigned to it, which substantially reduces the possibility of resource competition. However, resource competition may still occur if some pods use resources more than they request[2]. Therefore, the ranking module also takes account of the resource competition problem. In lines 8–9 of Algorithm 2, when a WN n_i already serves some pods (i.e., $\zeta_i > 0$), in addition to the processing delay $\tilde{D}_{\text{p}}(u_k, p_j, n_i)$ and the network delay $\tilde{D}_{\text{n}}(u_k, p_j, n_i)$ for n_i to handle a new pod p_j, we consider the extra delay Γ_i due to the context switch and resource competition:

$$\Gamma_i = \beta_{\text{CS}} + 2^{(\zeta_i - 1)}\beta_{\text{RC}}, \tag{18}$$

where β_{CS} is the delay caused by a context switch, which in general takes from 100 nanoseconds to some microseconds, depending on the CPU's architecture and the context's size [33]. The term $2^{(\zeta_i-1)}\beta_{\text{RC}}$ is the delay caused by the resource competition. According to [34], this delay grows exponentially with the number ζ_i of pods (on the scale of several milliseconds). Hence, we suggest setting β_{RC} to a few microseconds. By using the extra delay Γ_i, when a WN serves more pods, the chance of assigning a new pod to it will decrease (as the overall delay $\Omega_{i,j}$ rises). In this way, the ranking module will not assign many pods to the same WN, thereby mitigating the impact of resource competition.

2. As discussed in Section 5.1, Kubernetes allows a user to specify demand c_j^{CPU} and limit l_j^{CPU} of CPU resources for his/her pod p_j, where $c_j^{\text{CPU}} \leq l_j^{\text{CPU}}$. Hence, p_j can use more than c_j^{CPU} (but below l_j^{CPU}) CPU resources. The same situation is also applied to memory resources.

TABLE 3: Software programs used to create a Kubernetes cluster and set up the experiment environment.

software program	version
Kubernetes	1.19.9
VMware Workstation	15.5.6 build-16341506
operating system	Ubuntu 18.04.5 LTS
Linux kernel	5.4.0-58-generic
Golang	gol.15.2
Flannel	0.13.0
MetalLB	0.8.1
Contour	1.16.0
Prometheus	2.27.1
Grafana	8.0.1
ApacheBench	2.3
stress-ng	0.09.25-1_amd64

6.2 Dynamic Environment

In a dynamic environment, users may change resource usage at runtime. To adapt to such an environment, we ameliorate the DACS algorithm as follows: Suppose that a user changes the demand vector of pod p_j from $\vec{C}_j = (c_j^{\texttt{CPU}}, c_j^{\texttt{MEM}}, c_j^{\texttt{NET}})$ to $\vec{C}_{j'} = (c_{j'}^{\texttt{CPU}}, c_{j'}^{\texttt{MEM}}, c_{j'}^{\texttt{NET}})$. Besides, p_j is currently assigned to WN n_i. Then, two cases are considered.

- $r_i^{\texttt{X}} \geq c_{j'}^{\texttt{X}} - c_j^{\texttt{X}}$, for $\texttt{X} \in \{\texttt{CPU}, \texttt{MEM}, \texttt{NET}\}$: This case implies that n_i has enough residual resources to handle changes in p_j's resource demands. Hence, we let p_j stay in n_i for execution and update $r_i^{\texttt{X}}$ by $r_i^{\texttt{X}} - (c_{j'}^{\texttt{X}} - c_j^{\texttt{X}})$.
- Otherwise: To reduce the risk of resource competition at n_i, we let p_j migrate to another WN, say, n_k if n_k has sufficient resources to meet p_j's new demands. In this case, we update the resource vectors of both n_i and n_k by $r_i^{\texttt{X}} = r_i^{\texttt{X}} + c_j^{\texttt{X}}$ and $r_k^{\texttt{X}} = r_k^{\texttt{X}} - c_{j'}^{\texttt{X}}$, respectively. However, if no such WN can be found, we pick a WN with the most residual resources (possibly n_i) and assign p_j to that WN.

The work [35] considers deep learning applications hosted on the cloud with a cluster of containers, where each learning model is placed in a container. The objective is to offer users the services with QoE (quality of experience) targets. Based on these targets, WNs dynamically adjust resource limits for their containers to optimize the overall performance. This inspires us to improve DACS in future work to schedule containers that have QoE targets or may dynamically change limits on resources (i.e., $l_j^{\texttt{CPU}}$ and $l_j^{\texttt{MEM}}$) by users.

7 Performance Evaluation

This section discusses how to construct a Kubernetes cluster and set up our experiment environment, gives scenario design and scheduling methods for comparison, and evaluates system performance in two scenarios with homogeneous and heterogeneous WNs in the Kubernetes cluster.

7.1 Kubernetes Cluster Creation and Environment Setup

For performance evaluation, we create a Kubernetes cluster by the VMware Workstation [9], which runs on a computer equipped with an Intel-i7 3.6 GHz CPU (which has 4 cores) and 32 GB RAM. The operating system is Ubuntu with the Linux kernel of version 5.4. The Kubernetes cluster consists of one master node and three WNs, which are virtual machines. Since the master node has to manage the whole cluster, it is given more resources, specifically, 2 *virtual CPUs (vCPUs)* and 8 GB RAM. Each WN is allocated with resources of 1 or 2 vCPUs and

TABLE 4: Average RTTs between nodes in scenario 1.

node	master	WN n_1	WN n_2	WN n_3
master	–	200 ms	150 ms	160 ms
WN n_1	200 ms	–	50 ms	60 ms
WN n_2	150 ms	50 ms	–	10 ms
WN n_3	160 ms	60 ms	10 ms	–
average	170.0 ms	103.3 ms	70.0 ms	76.7 ms

TABLE 5: Average RTTs between nodes in scenario 2.

node	master	WN n_1	WN n_2	WN n_3
master	–	350 ms	210 ms	200 ms
WN n_1	350 ms	–	160 ms	150 ms
WN n_2	210 ms	160 ms	–	10 ms
WN n_3	200 ms	150 ms	10 ms	–
average	253.3 ms	220.0 ms	126.7 ms	120.0 ms

3 GB or 4 GB RAM, depending on the cloud or edge scenarios. The scheduler's program (to implement different scheduling methods) is written in the Go programming language (Golang) [36]. The detail of scenarios and scheduling methods will be discussed later in Section 7.2. To let nodes communicate with each other, we employ Flannel [37] to be the container network interface and build an IPv4 network to realize Kubernetes networking mentioned in Section 2.3.

We adopt both MetalLB [38] and Contour [39] to help users access their pods in the Kubernetes cluster. Whenever a user issues a service request, MetalLB picks an address from the predefined IP pool for the service. Afterward, Contour acts as a reverse proxy between users and pods. In this way, users need to know only the IP address of the reverse proxy, thereby facilitating the deployment of pods (based on the scheduling result). Moreover, since users do not know where their pods are assigned, the security strength can be improved.

The traffic control module of Linux is used to generate the transmission latency, which affects the network delay. We set the RTTs between nodes in the Kubernetes cluster according to both [24] and [40]. In particular, we consider that the master node is placed in the cloud (with higher latency), while WNs are IoT edge nodes with lower latency.

Moreover, we use Prometheus [41] to monitor the resource utilization of WNs, which collects data based on the HTTP pull method and stores data in a time series database. The collected data can be visualized through Grafana [42]. Besides, after deploying pods, we adopt ApacheBench [43] to measure the average network delay that users access their pods.

Regarding stress testing, we use the stress-ng (stress next generation) tool [44] to generate CPU workloads. The tool can produce a static workload for CPUs, like floating point, integer, and bit manipulation. Moreover, stress-ng also supports CPU bomb applications with dynamic workloads, such as the *fast Fourier transform (FFT)* [45]. To evaluate the effect of dynamic CPU workloads on different scheduling methods, we choose to use FFT and consider two cases. In the *low-load case*, each pod runs 250 FFT iterations. In the *high-load case*, each pod executes 3000 FFT iterations. Table 3 summarizes the software programs and their versions used to construct a Kubernetes cluster and set up the experiment environment.

7.2 Scenario Design and Comparing Methods

Two scenarios are designed to evaluate system performance in different environments. Scenario 1 is for a cloud environment, where all WNs are homogeneous in terms of resources. In particular, each WN is given 2 vCPUs and 4 GB RAM. There are 10 pods to be deployed, and Table 4 shows the average RTT between any two nodes in the Kubernetes cluster. On the other hand, scenario 2 considers an edge environment. WNs are IoT edge nodes, which are heterogeneous and with relatively fewer resources. More concretely, WNs n_1, n_2, and n_3 are allocated with 2 vCPUs, 1 vCPU, and 1 vCPU, and 4 GB, 4 GB, and 3 GB RAM, respectively. Since WNs have fewer resources, we reduce the number of pods to 6. Table 5 lists the average RTT between any two nodes in scenario 2.

The master node does not partake in the execution of user applications, so pods are merely assigned to WNs. To observe the effect of RTTs between the master node and WNs on system performance, we let some users login from the master node to access their pods (in this case, these pods are remote). The numbers of logged-in users and service requests of each node will be similar.

In addition to the kube-scheduler[3] (i.e., the default scheduler in Kubernetes), we compare our DACS algorithm with four scheduling methods discussed in Section 3:

- *ElasticFog* [24]: Based on the amount of network traffic, it assigns a corresponding number of pods to each WN. Every WN has at least one pod.
- *Task scheduling and resource allocation (TSRA)* [27]: By considering the execution time of each task and the transmission time for the task's data and container image, TSRA aims to maximize the number of tasks that can be completed before deadlines.
- *Network-aware scheduling (NAS)* [29]: Some WNs are designated as *preferred nodes (PNs)*. Pods are assigned to PNs if they have enough resources. Otherwise, NAS picks the WNs whose links to PNs have shorter RTTs.
- *Kubernetes container scheduling strategy (KCSS)* [20]: It places pods on WNs to raise resource utilization.

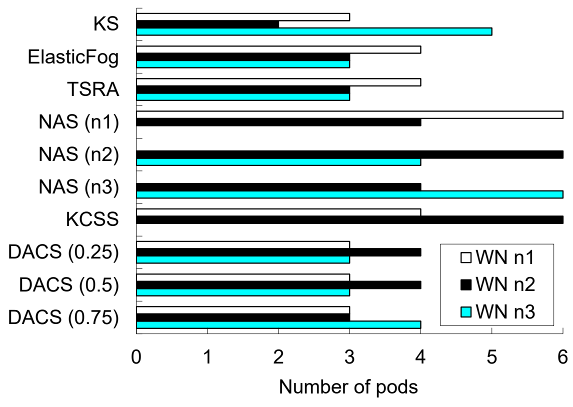
In DACS, we set α to 0.25, 0.5, and 0.75 to study its effect.

7.3 Scenario 1 (Cloud): Homogeneous WNs

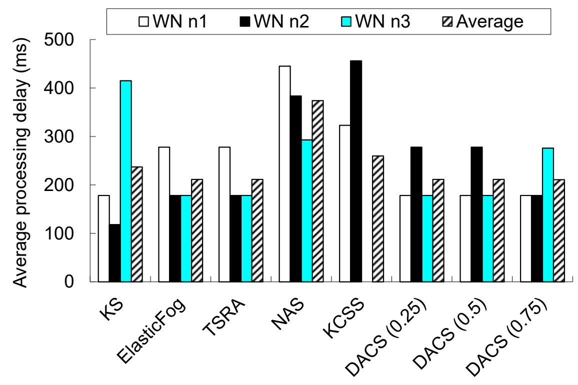
Scenario 1 is for a cloud environment, where WNs have the same number of resources. Fig. 3(a) shows the number of pods assigned to each WN by different methods. Specifically, the kube-scheduler scores WNs based on their residual resources. After allocating some pods to WNs n_1 and n_2, WN n_3 has the most residual resources. Thus, the remaining 5 pods are given to n_3. Regarding ElasticFog and TSRA, since all WNs are homogeneous, they allot 3 pods to each WN. The extra one pod is randomly assigned (here, they select n_1 to serve this pod). In NAS, the pod assignment depends on the choice of PNs. The PN (indicated in the brackets) handles 6 pods, while its neighbor with the minimum average RTT takes 4 pods. KCSS aims to improve resource utilization, so it places 4, 6, and 0 pods on n_1, n_2, and n_3, respectively. Unlike NAS and KCSS, DACS gives each WN 3 pods to balance their loads (as they have the same number of resources). When $\alpha \leq 0.5$, the proportion of network delay in Eq. (3) becomes relatively larger. In this case, DACS prefers to lower the network delay. That is why DACS chooses n_2 (with the minimum average RTT in Table 4) to handle the extra pod.

Fig. 3(b) and (c) present the average processing delays in the low-load and high-load cases, where each pod runs 250 and 3000 FFT iterations, respectively. The processing delay

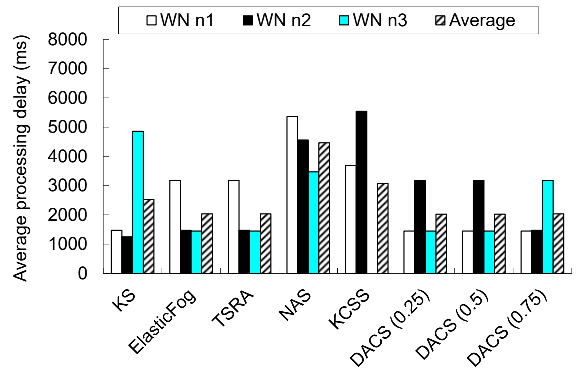
3. In Fig. 3 and 4, we mark the kube-scheduler as KS as an abbreviation.



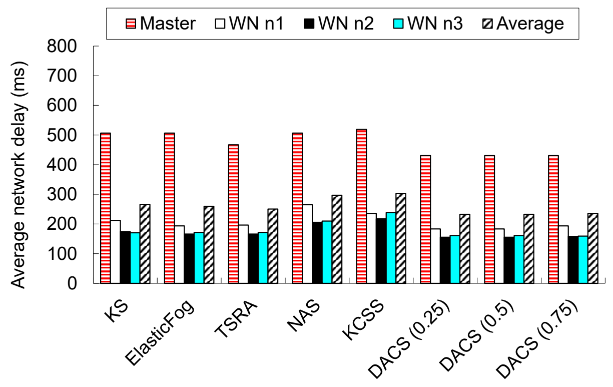
(a) pod assignment



(b) processing delay (low-load case)



(c) processing delay (high-load case)



(d) network delay

Fig. 3: Performance evaluation in scenario 1.

substantially rises in the high-load case, but the trend in each method does not change. Since the kube-scheduler allocates a half of pods to n_3, n_3 has a much higher processing delay than n_1 and n_2. For ElasticFog and TSRA, since n_1 handles one more pods, it has a higher processing delay than the other two WNs. Regarding NAS, we take the average of experimental results by evenly choosing n_1, n_2, and n_3 to be the PN. Since the PN has to handle 6 pods, it incurs a significantly high processing delay. In KCSS, n_1 and n_2 are assigned with 4 and 6 pods, respectively, so n_2 has a higher processing delay than n_1. As n_3 need not handle any pod, it has no processing delay. Like ElasticFog and TSRA, DACS assigns 3 pods to each WN and selects a WN (based on its average RTT) to handle one more pod, so the selected WN has a higher processing delay. From Fig. 3(b) and (c), ElasticFog, TSRA, and DACS have the lowest average processing delays, since they balance loads among WNs in a cloud environment with homogeneous WNs.

TABLE 6: Amount of time to complete all tasks in scenario 1.

method	low load	high load
kube-scheduler	13.25 s	127.00 s
ElasticFog	11.80 s	104.03 s
TSRA	11.78 s	104.01 s
NAS	16.62 s	198.51 s
KCSS	17.04 s	204.19 s
DACS (0.25)	11.31 s	101.52 s
DACS (0.5)	11.23 s	100.91 s
DACS (0.75)	11.02 s	99.41 s

Then, we evaluate the average network delay, as shown in Fig. 3(d). According to Table 4, the master node has higher RTTs, so users who login to the master node will encounter higher network delays. The kube-scheduler, ElasticFog, NAS[4], and KCSS methods do not consider the time spent to transmit data and container images of tasks. Hence, they have higher network delays. By taking the transmission time into account, TSRA has a lower network delay than the above methods. Our DACS algorithm uses Eq. (11) additionally to calculate network delays for remote pods, which considers the average RTT between two nodes. Thus, DACS can further reduce the average network delay, as compared with TSRA.

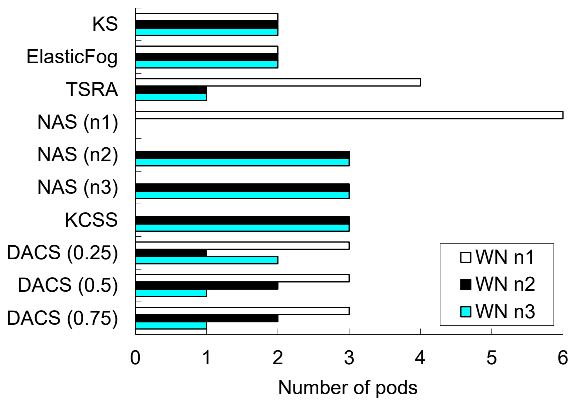
Table 6 lists the amount of time consumed by each method to complete all tasks in scenario 1. Our DACS algorithm has the minimum completion time when $\alpha = 0.75$. In the low-load case, DACS has 16.8%, 6.6%, 6.5%, 33.7%, and 35.3% less completion time than the kube-scheduler, ElasticFog, TSRA, NAS, and KCSS methods, respectively. In the high-load case, DACS saves 21.7%, 4.4%, 4.4%, 49.4%, and 51.3% of completion time, as compared with kube-scheduler, ElasticFog, TSRA, NAS, and KCSS, respectively.

7.4 Scenario 2 (Edge): Heterogeneous WNs

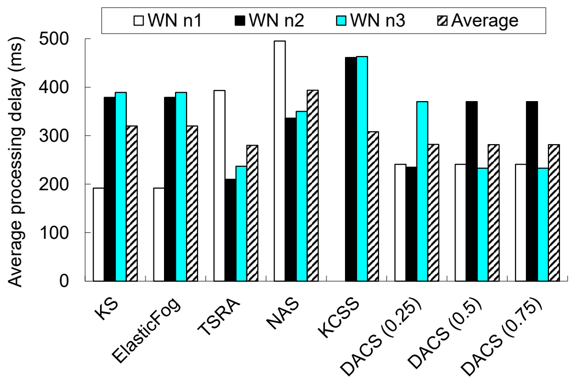
In scenario 2, we consider an edge environment, where WNs n_1, n_2, and n_3 are IoT edge nodes with different CPU and memory resources. In particular, n_1 has more CPU resources than n_2 and n_3, and n_3 is allocated with the fewest memory resources. Besides, the master node is located in the cloud, and the average RTTs between nodes are larger than in scenario 1 (referring to Tables 4 and 5).

Fig. 4(a) gives the number of pods assigned to each WN. The kube-scheduler first places pods on n_1 (with the most resources). After placing 2 pods, n_1 has fewer residual resources than others. Hence, the remaining pods are split between n_2

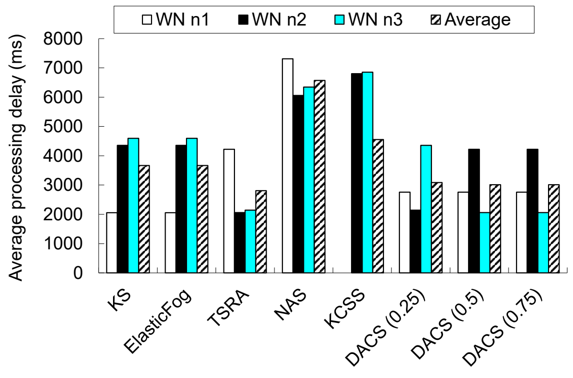
4. For NAS, it considers RTTs only when the PN can no longer serve pods (due to running out of resources).



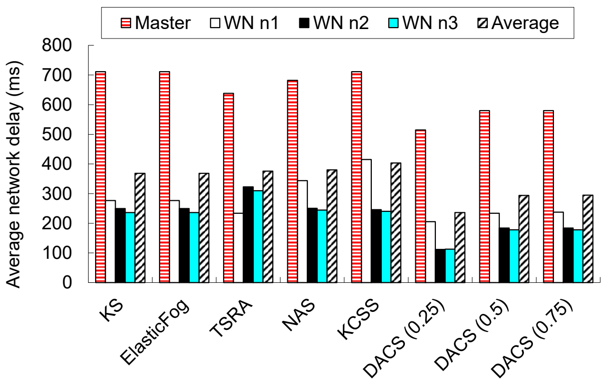
(a) pod assignment



(b) processing delay (low-load case)



(c) processing delay (high-load case)



(d) network delay

Fig. 4: Performance evaluation in scenario 2.

and n_3. For ElasticFog, since the amount of traffic of WNs is similar, each WN is assigned with 2 pods. TSRA prioritize picking a WN with adequate resources to serve pods (to meet deadlines). Therefore, TSRA gives 4, 1, and 1 pod to n_1, n_2, and n_3, respectively. In NAS, if n_1 is the PN, all pods are assigned to n_1 (as it has the most resources). Otherwise, n_2 and n_3 are each given 3 pods since each of them does not have enough resources to handle all pods. KCSS aims to maximize resource utilization, so it prefers placing pods on WNs with relatively fewer resources (i.e., n_2 and n_3). DACS assigns 3 pods to n_1, as n_1 has ample resources. If $\alpha = 0.25$, DACS prioritizes the network delay, so n_3 is given one more pod than n_2 (though they have the same number of CPU resources). When $\alpha \geq 0.5$, since n_2 has more memory resources than n_3, n_2 is assigned with more pods than n_3 instead.

TABLE 7: Amount of time to complete all tasks in scenario 2.

method	low load	high load
kube-scheduler	16.92 s	168.47 s
ElasticFog	15.95 s	160.13 s
TSRA	15.65 s	159.09 s
NAS	17.49 s	277.20 s
KCSS	19.59 s	284.81 s
DACS (0.25)	14.05 s	139.95 s
DACS (0.5)	14.37 s	141.05 s
DACS (0.75)	14.39 s	143.17 s

Fig. 4(b) and (c) present the average processing delays in both low-load and high-load cases, respectively. Since kube-scheduler and ElasticFog have the identical assignment of pods in Fig. 4(a), their average processing delays will be similar. Specifically, since n_1 has twice of CPU resources as n_2 and n_3, and each WN is given an equal number of pods, n_1's processing delay is nearly a half of that of n_2 and n_3. TSRA assigns 4 pods to n_1, making n_1's processing delay higher than others. In NAS, when n_1 is the PN, it has to cope with all pods. That is why n_1's processing delay is the highest in NAS. KCSS allots 3 pods to each of n_2 and n_3 (i.e., the same with NAS when n_2 and n_3 are the PNs). Thus, the processing delays of n_2 and n_3 are close to each other in KCSS and NAS. Our DACS algorithm gives 3 pods to n_1. Then, n_2 is given 1 or 2 pods, depending on α, and n_3 has to handle the remaining pods. Since n_1 has 2 vCPUs but either n_2 or n_3 has only 1 vCPU, n_1's processing delay is lower than the higher processing delay of n_2 and n_3.

Fig. 4(d) compares the average network delay. As discussed in Section 7.3, since TSRA considers the transmission time for data and container images of tasks, its average network delay will be lower than kube-scheduler, ElasticFog, NAS, and KCSS. Our DACS algorithm further takes account of RTTs between nodes, so DACS has the lowest network delay among all methods, especially when $\alpha = 0.25$. That is because the proportion of network delay is larger than that of processing delay in the objective function in Eq. (3).

Table 7 shows the amount of time spent by each method to wind up all tasks in scenario 2. DACS has the minimum completion time when $\alpha = 0.25$. In the low-load case, DACS decreases 17.0%, 11.9%, 10.2%, 19.7%, and 28.3% completion time than kube-scheduler, ElasticFog, TSRA, NAS, and KCSS, respectively. Regarding the high-load case, DACS reduces 16.9%, 12.6%, 12.0%, 49.5%, and 50.9% of completion time, as compared with the kube-scheduler, ElasticFog, TSRA, NAS, and KCSS methods, respectively. This result shows the superiority of our DACS algorithm in terms of task completion time in an edge environment.

8 Conclusion and Future Work

Kubernetes provides the orchestration and management of containerized applications and serves as a good platform for supporting edge-computing microservices. How to efficiently allocate pods to WNs plays a key role in Kubernetes performance. The kube-scheduler in Kubernetes is developed for a cloud environment with homogeneous nodes. However, IoT edge nodes have different resources and bandwidth. To this end, the paper proposes the DACS algorithm that consists of scheduling and ranking modules. For each pod, the scheduling module screens out WNs without sufficient resources to deal with this pod. Then, among those WNs that pass the screening, the ranking module picks a WN that not only has a shorter average delay but also holds more resources to serve the pod. By adopting the VMware Workstation to build a Kubernetes cluster for performance evaluation, we demonstrate that DACS strikes a good balance between processing delays and network delays among pods. Hence, DACS takes less time to complete tasks in cloud and edge environments, as compared with kube-scheduler, ElasticFog, TSRA, NAS, and KCSS.

For future work, we expect to deploy a number of IoT edge nodes on the cloud and then implement a Kubernetes cluster to manage these nodes. It is interesting to study how the deployment and applications of IoT edge nodes will affect Kubernetes performance and schedule containers accordingly to improve the performance. Moreover, as mentioned in Section 6.2, we will consider scheduling containers that have QoE targets or may dynamically change resource limits by users.

References

[1] Q. Luo, S. Hu, C. Li, G. Li, and W. Shi, "Resource scheduling in edge computing: a survey," *IEEE Comm. Surveys & Tutorials*, vol. 23, no. 4, pp. 2131–2165, 2021.

[2] Kubernetes. [Online]. Available: https://kubernetes.io/

[3] L. Toka, G. Dobreff, B. Fodor, and B. Sonkoly, "Machine learning-based scaling management for Kubernetes edge clusters," *IEEE Trans. Network and Service Management*, vol. 18, no. 1, pp. 958–972, 2021.

[4] T. Goethals, F.D. Turck, and B. Volckaert, "Extending Kubernetes clusters to low-resource edge devices using virtual kubelets," *IEEE Trans. Cloud Computing*, vol. 10, no. 4, pp. 2623–2636, 2022.

[5] L.H. Phuc, L.A. Phan, and T. Kim, "Traffic-aware horizontal pod autoscaler in Kubernetes-based edge computing infrastructure," *IEEE Access*, vol. 10, pp. 18966–18977, 2022.

[6] Z. Wan, Z. Zhang, R. Yin, and G. Yu, "KFIML: Kubernetes-based fog computing IoT platform for online machine learning," *IEEE Internet of Things J.*, vol. 9, no. 19, pp. 19463–19476, 2022.

[7] J. Park, U. Choi, S. Kum, J. Moon, and K. Lee, "Accelerator-aware Kubernetes scheduler for DNN tasks on edge computing environment," *Proc. IEEE/ACM Symp. Edge Computing*, 2021, pp. 438–440.

[8] LXC. [Online]. Available: https://linuxcontainers.org/

[9] VMware. [Online]. Available: https://www.vmware.com/

[10] S. Pakdel and A.C. Elster, "Using heterogeneous graph nodes (HGNs) to minimize overall graph execution time in heterogeneous distributed systems modeling," *Proc. Euromicro Int'l Conf. Parallel, Distributed and Network-Based Processing*, 2019, pp. 211–216.

[11] W. Fan, J. Zhu, and K. Ding, "An improved task duplication based clustering algorithm for DAG task scheduling in heterogenous and distributed systems," *Proc. IEEE Int'l Conf. Systems, Man, and Cybernetics*, 2022, pp. 878–883.

[12] Z. Tang, L. Du, X. Zhang, L. Yang, and K. Li, "AEML: an acceleration engine for multi-GPU load-balancing in distributed heterogeneous environment," *IEEE Trans. Computers*, vol. 71, no. 6, pp. 1344–1357, 2022.

[13] L. Richardson and S. Ruby, *RESTful Web Services*. Sebastopol: O'Reilly, 2007.

[14] Y.C. Wang and Y.C. Tseng, "Packet fair queuing algorithms for wireless networks," in *Design and Analysis of Wireless Networks*. Hauppauge: Nova Science Publishers, 2005, ch. 7, pp. 113–128.

[15] Calico. [Online]. Available: https://projectcalico.docs.tigera.io/reference/architecture/overview

[16] G.N. Purdy, *Linux Iptables Pocket Reference*. Sebastopol: O'Reilly, 2004.

[17] V. Medel, C. Tolon, U. Arronategui, R. Tolosana-Calasanz, J.A. Banares, and O.F. Rana, "Client-side scheduling based on application characterization on Kubernetes," *Proc. Int'l Conf. the Economics of Grids, Clouds, Systems, and Services*, 2017, pp. 162–176.

[18] C. Wobker, A. Seitz, H. Mueller, and B. Bruegge, "Fogernetes: deployment and management of fog computing applications," *Proc. IEEE/IFIP Network Operations and Management Symp.*, 2018, pp. 1–7.

[19] Y. Hu, C.D. Laat, and Z. Zhao, "Multi-objective container deployment on heterogeneous clusters," *Proc. IEEE/ACM Int'l Symp. Cluster, Cloud and Grid Computing*, 2019, pp. 592–599.

[20] T. Menouer, "KCSS: Kubernetes container scheduling strategy," *J. Supercomputing*, vol. 77, pp. 4267–4293, 2021.

[21] Z. Wang, H. Liu, L. Han, L. Huang, and K. Wang, "Research and implementation of scheduling strategy in Kubernetes for computer science laboratory in universities," *Information*, vol. 12, no. 1, pp. 1–10, 2021.

[22] J. Han, Y. Hong, and J. Kim, "Refining microservices placement employing workload profiling over multiple Kubernetes clusters," *IEEE Access*, vol. 8, pp. 192543–192556, 2020.

[23] D. Zhao, M. Mohamed, and H. Ludwig, "Locality-aware scheduling for containers in cloud computing," *IEEE Trans. Cloud Computing*, vol. 8, no. 2, pp. 635–646, 2020.

[24] 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.

[25] Y. Fu, S. Zhang, J. Terrero, Y. Mao, G. Liu, S. Li, and D. Tao, "Progress-based container scheduling for short-lived applications in a Kubernetes cluster," *Proc. IEEE Int'l Conf. Big Data*, 2019, pp. 278–287.

[26] H.M. Fard, R. Prodan, and F. Wolf, "Dynamic multi-objective scheduling of microservices in the cloud," *Proc. IEEE/ACM Int'l Conf. Utility and Cloud Computing*, 2020, pp. 386–393.

[27] L. Yin, J. Luo, and H. Luo, "Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing," *IEEE Trans. Industrial Informatics*, vol. 14, no. 10, pp. 4712–4721, 2018.

[28] R. Mahmud, A.N. Toosi, K. Ramamohanarao, and R. Buyya, "Context-aware placement of industry 4.0 applications in fog computing environments," *IEEE Trans. Industrial Informatics*, vol. 16, no. 11, pp. 7004–7013, 2020.

[29] J. Santos, T. Wauters, B. Volckaert, and F.D. Turck, "Resource provisioning in fog computing: from theory to practice," *Sensors*, vol. 19, no. 10, pp. 1–25, 2019.

[30] J. Santos, T. Wauters, B. Volckaert, and F.D. Turck, "Towards delay-aware container-based service function chaining in fog computing," *Proc. IEEE/IFIP Network Operations and Management Symp.*, 2020, pp. 1–9.

[31] Kubernetes Documentation, "Resource management for pods and containers." [Online]. Available: https://kubernetes.io/docs/concepts/configuration/manage-resources-containers/

[32] Kubernetes Documentation, "Resource quotas." [Online]. Available: https://kubernetes.io/docs/concepts/policy/resource-quotas/

[33] A. Silberschatz, G. Gagne, and P.B. Galvin, *Operating System Concepts*. Hoboken: Wiley, 2019.

[34] E. Kim, K. Lee, and C. Yoo, "On the resource management of Kubernetes," *Proc. IEEE Int'l Conf. Information Networking*, 2021, pp. 154–158.

[35] Y. Mao, W. Yan, Y. Song, Y. Zeng, M. Chen, L. Cheng, and Q. Liu, "Differentiate quality of experience scheduling for deep learning inferences with docker containers in the cloud," *IEEE Trans. Cloud Computing*, pp. 1–11, 2023.

[36] The Go Programming Language. [Online]. Available: https://go.dev/

[37] Flannel. [Online]. Available: https://github.com/flannel-io/flannel

[38] MetalLB. [Online]. Available: https://metallb.universe.tf/

[39] Contour. [Online]. Available: https://projectcontour.io/

[40] A.J. Fahs and G. Pierre, "Proximity-aware traffic routing in distributed fog computing platforms," *Proc. IEEE/ACM Int'l Symp. Cluster, Cloud and Grid Computing*, 2019, pp. 478–487.

[41] Prometheus. [Online]. Available: https://prometheus.io/

[42] Grafana. [Online]. Available: https://grafana.com/

[43] ApacheBench. [Online]. Available: https://httpd.apache.org/docs/2.4/programs/ab.html

[44] Stress-ng. [Online]. Available: https://wiki.ubuntu.com/Kernel/Reference/stress-ng

[45] M.G. Xavier, K.J. Matteussi, F. Lorenzo, and C.A.F.D. Rose, "Understanding performance interference in multi-tenant cloud databases and web applications," *Proc. IEEE Int'l Conf. Big Data*, 2016, pp. 2847–2852.