Kubernetes in Fog Computing: Feasibility Demonstration, Limitations and Improvement Scope

(Invited Paper)

Paridhika Kayal
Department of Electrical and Computer Engineering,
University of Toronto
Email: paridhika.kayal@mail.utoronto.ca

***Abstract*—Fog computing (also known as edge computing) is a decentralized computing architecture that seeks to minimize service latency and average response time in IoT applications by providing compute and network services physically close to end-users. Fog environment consists of a network of *fog nodes* and IoT applications are composed of containerized *microservices* communicating with each other. Due to limited resources of fog nodes, it is often not possible to deploy all the containers of an application on a single fog node. Therefore, communicating containers need to be distributed on multiple fog nodes. Distribution and management of containerized IoT applications is always a critical issue to the system performance in a fog environment. Kubernetes, an open-source system, has grown into a container orchestration standard by simplifying the deployment and management of containerized applications. Despite the progress made by the academia and industry with respect to container management and the wide-scale acceptance of Kubernetes in cloud environments, container management in fog environment is still in the early stage in terms of research and practical deployment. This article aims to fill this gap by analyzing the expediency of Kubernetes container orchestration tool in the fog computing model. The paper also highlights limitations with the current Kubernetes approach and provide ideas for further research to adapt to the needs of the fog environment. Lastly, we provide experiments that demonstrate the feasibility and industrial practicality of deploying and managing containerized IoT applications in the fog computing environment.**

***Index Terms*—Fog computing, fog nodes, microservices, Kubernetes, Docker.**

I. Introduction

According to the OpenFog consortium [1], fog computing is defined as a system-level horizontal architecture that distributes services by dynamic pooling of unused local resources from participating end-user devices which are referred to as *fog nodes*. Fog nodes are the computational, networking, storage and acceleration devices in fog computing. Fog nodes establish a communication network, which we refer to as *fog network*, where each fog node is connected to some other fog nodes by wireless or wired links. Containerization enables developers to create distributed IoT applications made up of small discrete pieces called *microservices*. Microservices talk to each other and can run across multiple servers, as opposed to the large monolithic applications running on a single server. Docker [2] is an open-source project that has emerged as the enabling technology to build and scale containerized IoT applications. The resource management system that allocates limited fog resources to containerized applications with diverse resource requirements is a crucial component of fog computing technology. This suggests the need for effective orchestration services for the deployment and management of containerized IoT applications.

Kubernetes [3], the Google-incubated open-source container orchestration tool, is quickly becoming the de facto standard for managing large container deployments in the cloud environment. Companies like Red Hat, Microsoft, IBM, Amazon, Mirantis and VMWare have integrated Kubernetes into their cloud platforms. Due to distributed nature, heterogeneity and limited resources of fog nodes, fog computing environment have different orchestration requirements and a well-justified approach is needed to tackle the challenges associated with the deployment and management of IoT applications on fog nodes. Thus, the analysis of the performance and usability of Kubernetes in the fog computing environment is an interesting and relatively new area of research. The primary goals of the present paper are as follows: (i) describing the fog computing model for the distribution of containerized IoT application; (ii) reviewing the architecture and implementation of the Kubernetes container orchestration service; (iii) pointing out the limitations of the current Kubernetes container orchestration design which have not fully received attention so far in the fog computing environment; (iv) offering suggestions for remedying these shortcomings to shape Kubernetes to be suited for the fog environment; and (v) demonstrating the feasibility and industrial practicality of deploying and managing containerized IoT applications in the fog computing environment. In our experiment, we deploy an IoT application on the network of raspberry pis acting as fog nodes. The deployment utilizes commercial software tools like Docker for container virtualization, Kubernetes as the container orchestration tool and Weave Net for creating overlay network.

The rest of the paper is organized as follows. We discuss the related work that exploits Kubernetes for the edge/fog environment in Sec. II. We describe the fog computing model in Sec. III and Kubernetes container orchestration architecture in Sec. IV. Limitations and improvement scope in existing Ku-

978-1-7281-5503-6/20/$31.00 ©2020 IEEE

bernetes architecture are discussed in Sec. V. We demonstrate the practicality of proposed model in Sec. VI and we conclude Sec. VII.

II. Related work

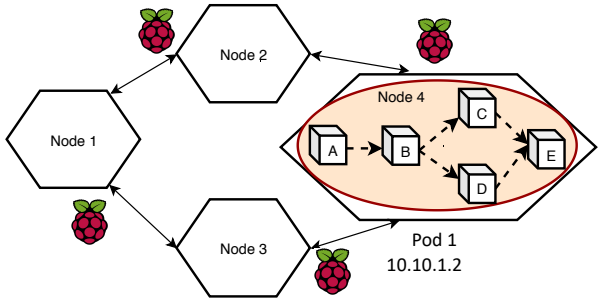
Most of the prior work that explores Kubernetes as a container orchestration tool mainly focuses on two key aspects: (i) optimizing the Kubernetes container orchestration system to make it suitable to run on resource-constrained edge devices; (ii) providing improvements to different Kubernetes modules including Kubernetes controller, scheduler, and networking model.

Various alternatives based on modifications in Kubernetes have been proposed to provide a lightweight orchestration system to run on devices. KubeEdge [4] is an open-source system built upon Kubernetes and provides core infrastructure support for networking, application deployment and metadata synchronization between cloud and edge. With KubeEdge, users can orchestrate apps, manage devices and monitor app and device status on Edge nodes just like a traditional Kubernetes cluster in the cloud. It also supports MQTT and allows developers to author custom logic and enable resource-constrained device communication at the edge. K3S [5] is a highly available, certified Kubernetes based container orchestrator, modified specifically for edge devices. It is designed for production workloads in unattended, resource-constrained, remote locations or inside IoT appliances. K3s provides support for multi-arch images and works great for something as small as a raspberry pi. MicroK8s [6] is a small, fast, single-package Kubernetes for developers, IoT, and edge. It is great for offline development, prototyping, and testing. FLEDGE [7] is yet another Kubernetes compatible edge container orchestrator to achieve low-resource container orchestration for resource-constrained edge devices. All these alternatives provide orchestration tools that are optimized to run on resource-constrained devices but do not deal with shortcomings of the orchestration method itself to meet the scheduling requirements for constrained environments.

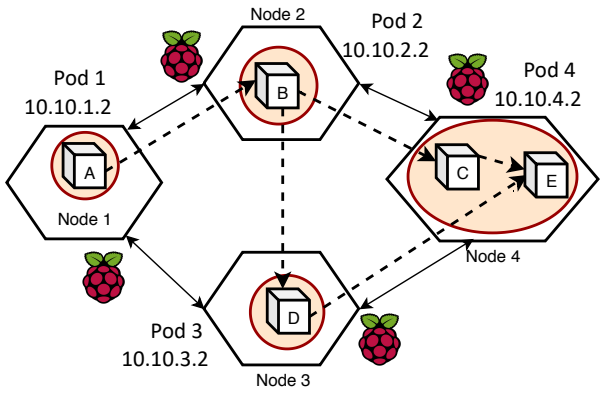
The existing literature has numerous papers focusing on the improvement in Kubernetes modules for cloud and fog-cloud platforms. In [8], the Kubernetes scheduling model is improved by combining the ant colony algorithm and particle swarm optimization algorithm. Authors in [9], presents a scalable and comprehensive controller for Kubernetes, KEIDS, in edge-cloud environments for IIoT with the view to minimize the carbon footprint rate while enhancing the performance and energy-efficiency. Another topology-based GPU scheduling framework based on the traditional Kubernetes GPU scheduling algorithm is proposed in [10]. In[11], a network-aware scheduling approach for container-based applications is proposed and validated using the Kubernetes platform. In [12] a custom scheduler for Kubernetes is proposed that delegates the scheduling decision among the processing nodes. The distribution of the scheduling task is achieved by transferring node filtering and node ranking jobs to a multipurpose Multi-Agent System. At present, the existing research work mainly improves the original Kubernetes model for cloud or fog-cloud environments. However, the use of Kubernetes in a fog-only environment is a new line of research. This paper fills the gap in this area by investigating the limitations of Kubernetes' default orchestration framework and proposing suggestions to adjust to the needs of the fog computing environment. Moreover, this paper demonstrates the practical feasibility of the deployment of IoT applications on a fog environment consisting of raspberry pi devices.

III. Fog Computing Model

In this section, we describe the fog computing model of distributing containerized IoT applications on the network of fog nodes. Figure 1 shows an example of IoT application consisting of five microservices, indicated by cubes, placed over a network of four fog nodes depicted by hexagons. The arrows between the containers indicate that they exchange data with each other. The bidirectional edges between the fog nodes indicate the communication links. The allocation of applications to the fog nodes in fog computing can be done in two ways as shown in Figure 1. Figure 1a shows the entire application placed on a single fog node, similar to the cloud deployments. This approach not only reduces latency but also improves the application QoS. However, in



(a) Application on a single fog node.



(b) Application distributed over fog network.

Fig. 1. Distribution of IoT Application.

a fog computing environment, due to limited resources, it is often not possible to deploy the entire application on a single fog node. Therefore, the application needs to be distributed over the network of communicating fog nodes as shown in Figure 1b. Deploying communicating containers on different fog nodes leads to data exchange between the fog nodes over the network, referred to as *communication cost*. The remaining of the paper aims to establish the feasibility of the proposed fog computing model using commercial tools and a practical IoT application.

IV. Kubernetes' Container Orchestration

Kubernetes [3], an open-source system, has grown into a container orchestration standard by simplifying the deployment and management of containerized applications. Kubernetes follows a master-slave type of architecture. The master node initializes and manages the cluster, and it is the entry point for all the administrative tasks. A slave/worker node joins the cluster and is controlled by the master node. Kubernetes nodes are physical devices running services like Docker which provides a container runtime and kubelet which executes containers on the node as dictated by the master node. *Pods* and *services* are two objects of Kubernetes that are essential to the comprehension of our work. A pod is the basic control and management unit of Kubernetes consisting of one or more containers, indicated by circles and ovals in Fig. 1. Kubernetes uses the "IP-per-pod" model to provide a unique IP address to the pods. Every container in a pod shares the network namespace, including the IP address and network ports. The pods are scheduled on the worker nodes, which have the necessary tools to run and connect them. A service is an abstraction that defines a logical set of pods and a policy to expose them as an application. The set of pods targeted by a service is usually determined by a selector.

Kubernetes orchestration algorithm consists of two major components: scheduling and networking.

A. Kube-Scheduler (KS): Kubernetes Default Scheduler Model

The resource scheduling of Kubernetes refers to the process of mapping *pods* on the available *nodes*; it is the key to solving the high availability and high reliability of large-scale industry tasks. The scheduler watches newly created pods or other unscheduled pods without a nodeName parameter. The nodeName is what shows which node should be owning this pod. The scheduler selects a suitable node for this pod through a set of rules and updates the pod definition with the node name. The kubelet on the chosen node is notified that there is a pod that is pending execution. The kubelet executes the pod, and the latter starts running on the node. The scheduling algorithm of Kubernetes mainly includes two main decision-making processes:

- Filtering: The scheduler uses a set of rules called *predicates* to filter out nodes that do not meet the requirements. For example, if the pod specifies the amount of resources required, then the scheduling algorithm filter the nodes based on *PodFitsResources* predicate. If the free amount of resources (CPU and memory) on a given node is less than the one required by the pod then the node is considered to fail the predicate and is excluded from the process. The rest of the nodes form a group of *feasible* nodes.
- Scoring: The scheduler tests each node that survived filtering against some functions that give it a score. For example, it tries to spread pods across nodes and zones while at the same time favoring the least loaded nodes (where "load" is measured as the sum of the resource requests of the containers running on the node, divided by the node's capacity). Finally, the node with the highest score is chosen (or, if there are multiple such nodes, then one of them is chosen at random).

B. Kubernetes Networking Model

Network communication between containers is the most difficult part because Kubernetes manages multiple nodes running several containers that need to communicate with each other. One of the important areas of managing the Kubernetes network is to forward container ports internally and externally to make sure containers and pods can communicate with one another properly. If the container's network communication is only within a single node, then Docker [2] network can be used to discover the peer. However, along with multiple nodes, Kubernetes uses an overlay network or container network interface (CNI) to achieve multiple container communication. To manage the communications between the containers and pods in Fig. 1, we require the following two Kubernetes networking models:

- Container-to-container communications: Containers from an application on the same node can be placed inside the same pod and can communicate with each other on localhost. (See Fig. 1a)
- Pod-to-pod communications: Containers on different nodes are placed in different pods as shown in Fig. 1b. We use weave net CNI plugin [13] to enable communication between the containers placed on different nodes. Weave

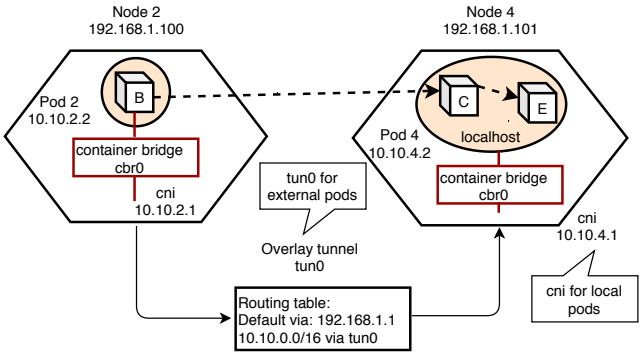


Fig. 2. Kubernetes networking for placement in Fig. 1b.

Net creates a virtual network that connects Docker containers across multiple hosts and enables their automatic discovery. Fig. 2 shows an example of the networking between the containers on Node 2 and Node 4 from Fig. 1b.

V. Limitations and Improvement Scope

In this section, we highlight the limitations of the current Kubernetes implementation that needs to be addressed in the future. We also propose suggestions to overcome these limitations. We identify the Kubernetes master-slave model and Kubernetes scheduling algorithm as two components of Kubernetes that have scope for improvement.

As described in Sec. IV, Kubernetes architecture has a master node which is a centralized component that runs an API server, a scheduler, and a controller manager to control and manage Kubernetes worker nodes. However, fog computing is a distributed computing paradigm specially designed to match the needs of the decentralized nature of IoT applications. Thus, the centralized nature of the Kubernetes orchestration system does not align well with the needs of the distributed fog computing environment. Therefore, modifying Kubernetes to remove centralized control is an area in need of exploration. There are potential scheduling strategies [14, 15] that do not involve a centralized controller and could be integrated with the default Kubernetes system.

Next, we point out the following three shortcomings of the KS algorithm that needs to be addressed and suggest a research agenda to deal with these issues.

1) The first disadvantage of the KS algorithm is that it does not consider the resource requests of all pods in the whole cluster. In other words, pod priority is not considered and a randomly selected pod is scheduled on the node with the highest node score, and hence the global optimal solution cannot be made.
2) Secondly, in fog computing, nodes have limited resources; hence it is highly probable that the filtering process (described in Sec. IV) is unable to find a feasible node that can fit the pod (as in Fig. 1a) and the pod is marked as unallocated. However, the containers in the pod can still be scheduled by spreading them across different nodes (see Fig. 1b).
3) Thirdly, although KS provides flexible and powerful features, the metrics applied in the decision-making process are rather limited. Only CPU and RAM usage rates are considered in the service scheduling while latency or bandwidth usage rates are not considered at all. Therefore, by using the Kubernetes default scheduling for the pods belonging to the same application, we may incur high communication costs if the communicating pods are scheduled on nodes that are far apart. Hence, the scoring algorithm needs to consider the topology of the fog network to minimize the communication cost between the communicating containers.

Kubernetes provides three ways to add new scheduling rules to the KS algorithm: (1) by adding new predicates or priorities to the scheduler and recompiling. (2) implementing a new scheduler that runs instead of, or alongside, the default scheduler and instruct Kubernetes what scheduler to use for each of the pods. (3) implementing ”scheduler extender” that the default KS calls out as a final step when making scheduling decisions. The third approach is particularly suitable for designing the strategy to make scheduling decisions in order to overcome the above-mentioned limitations. Kubernetes scheduling framework provides a new set of ”plugin” APIs to be added to the existing Kubernetes scheduler. Plugins are compiled into the scheduler, and these APIs allow many scheduling features to be implemented as plugins while keeping the scheduling ”core” simple and maintainable. In order to overcome the identified shortcomings, Figure 3 depicts our proposed improvements. We propose the following three plugin extensions to implement the required functionality.

1) Queue Sort Plugin: This plugin can be used to sort pods in the scheduling queue in decreasing order of the degree of communication between the containers of the application. With this sorting, pods with more communication between its containers will be scheduled first. The queue sort plugin provides ”less(pod1, pod2) bool” which can be used to find the pod with more communication between its containers.

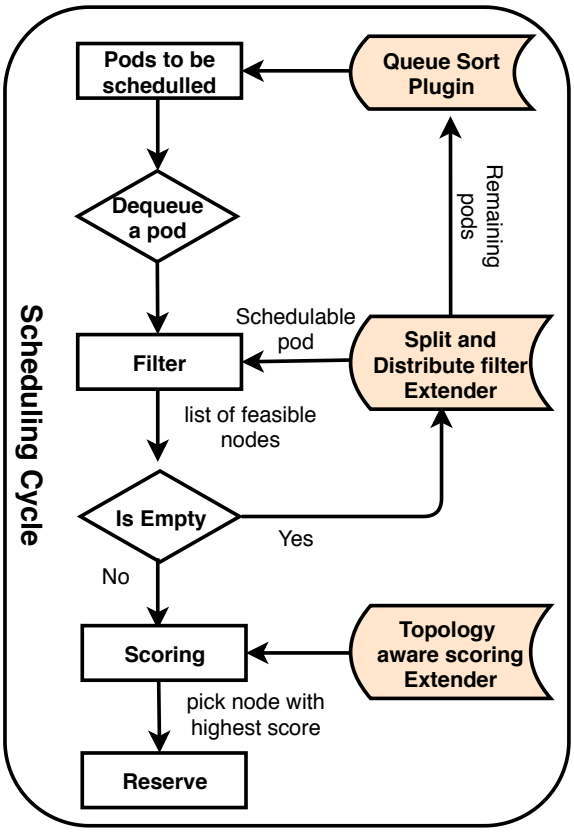


Fig. 3. Proposed scheduling design for fog computing platform.

2) Split and Distribute Extender Plugin: The split and distribute extender plugin would watch the pods for which no feasible node is found after the filtering step. The extender then kills the current pod and splits the pod into two pods such that the KS finds a feasible node for one of the two pods. We propose to use a greedy algorithm from [16, Algorithm 1] to find a group of maximum communicating containers that can be scheduled on one of the available nodes. The two pods then can be wrapped into a service that enables loose coupling between the dependent pods. Pods belonging to the same service are identified by the labels. Figure 4 shows an illustration of this process. Next, the pod for which KS does not find a feasible node can be moved back to the queue of unscheduled pods.

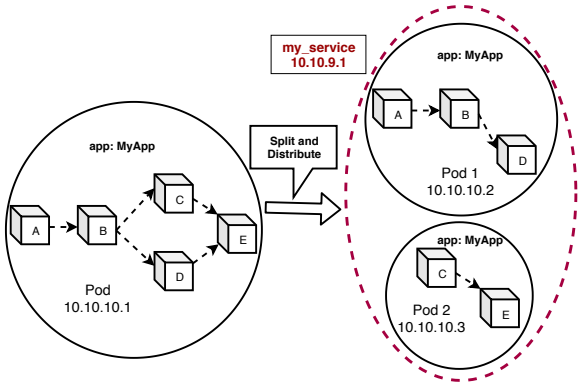


Fig. 4. Split and Distribute Visualization.

3) Topology Aware Scoring Extender Plugin: The scoring provides a lot of flexibility to integrate advanced features to the scheduling algorithm. We propose a topology-aware extender for the scoring function that prioritizes the nodes depending on their distances to reduce the communication cost between communicating containers. This approach has been previously proposed in [16] and has been found effective in reducing communication costs. The extender would then prefer to place pods belonging to the same service on the nodes in the same zone or region.
Kubernetes provide *MatchNodeSelector (affinity/anti-affinity)* feature that can be utilized to implement topology-aware scoring. By using node selectors (labels), it is possible to define that a given pod can only run on a particular set of nodes with an exact label value (node-affinity), or even that a pod should avoid being allocated on a node that already has certain pods deployed (pod-anti-affinity). Essentially, affinity/anti-affinity rules are properties of pods that attract/repel them to a set of nodes or pods. The rules are of the form "this pod should (or, in the case of anti-affinity, should not) run in an X if that X is already running one or more pods that meet rule Y".

Our vision is that this approach will provide a lot of flexibility to the application developers by supporting the deployment of a wide variety of IoT applications. Instead of splitting their large containerized applications into pods connected through Kubernetes services, they can simply provide a fully containerized application and get the applications deployed in the fog environment, ready to use. Next, we describe our experiments that provide validation for the feasibility of the proposed model.

VI. Experiments

We use real physical devices to implement the master-slave model of Kubernetes. Kubernetes master node initializes a cluster and generates a token that is used by worker nodes to join the cluster. Our testbed consists of a PC (Intel x86 architecture) serving as Kubernetes master node and four raspberry pi 3 (32-bit arm architecture) acting as worker nodes and forming a network as shown in Figure 1. Feasibility of using raspberry pi devices to implement a fog network and deploying Docker containers for IoT applications has previously been established by author's in [17]. All devices have Ubuntu 18.04 LTS as the operating system and run Kubernetes v1.17.3 and Docker-engine v19.03.6. The installation instructions for configuring raspberry pis are available on GitHub at [18]. We use a nodeJS IoT application (node v13.5.0) that requires real-time low latency services to show the practicality and industrial applicability of the proposed model. Kubernetes provides tools that automate the distribution of applications across a cluster of machines. Next, we configure Kubernetes to deploy the application according to the two placement strategies shown in Figure 1 for conducting experiments and to compare the performance results in the two scenarios. Each microservice of the application is containerized as a multi-platform image using Docker Buildx CLI plugin and pushed to the Docker hub repository.

A Kubernetes pod is defined as a YAML file that consists of the details of one or more container images. Listing 1 shows an example of pod created for microservice A in Figure 1. For the deployment strategy depicted in Figure 1a, we place all the Docker containers of the application in the same pod. The Kubernetes master node then deploys the pod on one of the raspberry pis. Every container in the pod communicates with one another using the localhost without any network delay. On the other hand, in order to implement the placement depicted in Figure 1b, the containers distributed on different Kubernetes nodes are placed in different pods. Kubernetes assigns a different IP address to each pod and communication between two pods placed on two different physical devices is performed over the network. We use Weave Net[13] to create a virtual network that connects Docker containers deployed across multiple nodes and enables their automatic discovery. This deployment strategy introduces a network delay in achieving the same functionality. We further use Kubernetes services of type "NodePort" that exposes the service on each node's IP at a static port (the NodePort).

This allows us to access the pod from outside the cluster, by requesting <nodeIP>:<NodePort>.

Listing 1
CREATING POD FOR MICROSERVICE A.

```
apiVersion: v1
kind: Pod
metadata:
  name: video-demo-sender-raspbian
  labels:
    purpose: demonstrate_streaming
spec:
  containers:
  - name: sender-demo-container-raspbian
    image: paridhika/testimages:video-
        sender-raspbian
    imagePullPolicy: Always
    ports:
      - name: video
        containerPort: 3000
        protocol: TCP
```

The experiment shows that the functionality of the application is not affected by the distribution of communicating containers in separate pods placed on different Kubernetes nodes. We further observe the quality of video streaming achieved with the two deployment strategies shown in Figure 1 and the time lag experienced in the second scenario (Figure 1b). This experiment establishes the practicality and industrial applicability of the proposed model for distributing IoT applications. It also establishes the future scope of integrating other user devices like smartphones and smart cameras as Kubernetes nodes.

VII. CONCLUSION

Fog computing has been regarded as an ideal platform for distributed and diverse IoT applications. Kubernetes, an open-source container orchestration tool, has become the standard of fact for deploying containerized applications in cloud and edge/fog computing environments. In this work, we first describe the fog computing model for the distribution of IoT applications and discuss the architecture of the Kubernetes container orchestration system. The use of Kubernetes for the deployment of applications in fog environment is under-researched and this paper tries to fill this gap. This paper also ventures to shape Kubernetes to be suited for the fog environment by proposing required extensions that open up new possibilities to move this research area further. We further provide a demonstration to show the feasibility of distributing communicating microservices on different raspberry pi devices acting as fog nodes. The experimental results show that even after splitting the application into multiple communicating pods distributed over the network, we can achieve the required functionality of the IoT application without significant penalty.

REFERENCES

[1] OpenFog Consortium Architecture Working Group, "OpenFog architecture overview white paper," 2017.

[2] "Docker Engine," 2020. [Online]. Available: https://docs.docker.com/engine/

[3] "Kubernetes: Production-Grade Container Orchestration," 2020. [Online]. Available: https://kubernetes.io/

[4] "KubeEdge, a Kubernetes Native Edge Computing Framework," 2020. [Online]. Available: https://kubeedge.io/

[5] "Rancher labs - k3s lightweight kubernetes." 2020. [Online]. Available: https://k3s.io/

[6] "MicroK8s: Zero-ops Kubernetes for workstations and edge/IoT," 2020. [Online]. Available: https://microk8s.io/

[7] T. Goethals, F. De Turck, and B. Volckaert, "Fledge: Kubernetes compatible container orchestration on low-resource edge devices," in *IoV. Technologies and Services Toward Smart Cities*, C.-H. Hsu, S. Kallel, K.-C. Lan, and Z. Zheng, Eds., 2020, pp. 174–189.

[8] Z. Wei-guo, M. Xi-lin, and Z. Jin-zhong, "Research on kubernetes' resource scheduling scheme," in *Proc. of ICCNS*, 2018, p. 144–148.

[9] 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 IoT-J*, Sep. 2019.

[10] S. Song, L. Deng, J. Gong, and H. Luo, "Gaia scheduler: A kubernetes-based scheduler framework," in *IEEE ISPA/IUCC/BDCloud/SocialCom/SustainCom*, Dec. 2018, pp. 252–259.

[11] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applications," in *IEEE NetSoft*, June 2019, pp. 351–359.

[12] O. Casquero, A. Armentia, I. Sarachaga, F. Pérez, D. Orive, and M. Marcos, "Distributed scheduling in kubernetes based on mas for fog-in-the-loop applications," in *IEEE ETFA*, Sep. 2019, pp. 1213–1217.

[13] "Weave Net addon for Kubernetes," 2020. [Online]. Available: https://www.weave.works/docs/net/latest/kubernetes/kube-addon/

[14] P. Kayal and J. Liebeherr, "Autonomic service placement in fog computing," in *IEEE WoWMoM*, June 2019, pp. 1–9.

[15] P. Kayal and J. Liebeherr, "Poster: Autonomic service placement in fog computing," in *Proc. of S3 Workshop at ACM MobiCom*, Oct. 2019.

[16] P. Kayal and J. Liebeherr, "Distributed service placement in fog computing: An iterative combinatorial auction approach," in *IEEE ICDCS*, July 2019, pp. 2145–2156.

[17] P. Bellavista and A. Zanni, "Feasibility of fog computing deployment based on docker containerization over raspberrypi," in *Proc. of ICDCN*, 2017.

[18] "Setup instructions," 2020. [Online]. Available: https://github.com/paridhika/streaming-app