Efficient Load-Balancing and Container Deployment for Enhancing Latency in an Edge Computing-Based IoT Network Using Kubernetes for Orchestration

Garrik Brel Jagho Mdemaya[1], Milliam Maxime Zekeng Ndadji[2],
Miguel Landry Foko Sindjoung[3], Mthulisi Velempini[4]
Department of Computer Science, University of Limpopo, Mankweng, South Africa[1,3,4]
Department of Mathematics and Computer Science, University of Dschang, Dschang, Cameroon[2]

***Abstract*—Edge Computing (EC) provides computational and storage resources close to data-generating devices, and reduces end-to-end latency for communications between end-devices and the remote servers. In smart cities (SC) for example, thousands of applications are running on edge servers, and it becomes crucial to manage resource allocation and load balancing to improve data transmission throughput and reduce latency. Kubernetes (k8s) is a widely used container orchestration platform that is commonly employed for the efficient management of containerized applications in SC. However, it does not integrate well with certain EC requirements such as network-related metrics and the heterogeneity of EC clusters. Furthermore, requests are equally distributed across all replicas of an application, which may increase the time taken for processing, since in the EC environment, nodes are geographically dispersed. Several existing studies have investigated this problem, unfortunately, the proposed solutions consume a lot of node's resources in the cluster. To the best of our knowledge, none of studies considered the cluster heterogeneity when deploying applications that have different resource requirements. To address this issue, this paper proposes a new technique to deploy applications on edge servers by extending Kubernetes scheduler, and an approach to manage requests among the different nodes. The simulation results show that our solution generates better results than some of the state-of-the-art works in terms of latency.**

Keywords*—*Latency; Kubernetes; edge computing; Internet of Things; load-balancing

I. Introduction

The fusion of Internet of Things (IoT) and edge computing has revolutionized distributed computing, enabling real-time data processing and analysis at the network edge in metropolitan cities [1, 11]. Various technologies are enhancing urban life. Numerous fundamental departments of cities such as transportation, power plants, information systems, crime detection traffic monitoring, etc. are equipped with sensors to gather data on community services. The endeavour to construct a smart city entails an increasing number of sensors and a growing volume of traffic. This evolution also impacts applications operating within smart city-related frameworks. Thus, thousands of applications are running in real-time on nodes at the edge layer. So they need to be organised well and orchestrated to serve efficiently each activity sector. Kubernetes (k8s), a leading container orchestration platform, has extended its reach to the edge, offering sophisticated tools for managing containerized applications in this decentralized environment [15, 3]. However, orchestrating containers on heterogeneous edge clusters poses significant challenges. Edge nodes come in diverse forms, ranging from resource-constrained sensors to powerful edge servers, each with distinct processing capabilities and network connectivity. Also, the edge layer is made up of several heterogeneous machines forming a cluster. Some machines are much more powerful than others; consequently, certain processes, such as artificial intelligence training, need to be run on much more powerful machines, while certain processes, such as saving and collecting multimedia data, can be run on less powerful machines.

Despite heterogeneity, smart deployment of applications and efficient utilization of computing resources require the distribution of incoming requests across machines within the cluster. These tasks fall under the kube-scheduler which is the component in charge of application deployment on nodes in a Kubernetes cluster, and under the ingress controller and kube-proxy components, which play a significant role in balancing the workload and optimizing performance across heterogeneous edge nodes. Several existing works designed to improve latency and throughput in edge computing-based IoT networks using Kubernetes as a container orchestration tool have been proposed in the literature. However, some of the proposed techniques designed to extend the default kube-scheduler consume a fair amount of energy due to the workload on nodes [12, 9]. Others use custom orchestrators running in containers in the same cluster [20, 7], which consumes the hardware resources of machines whose main aim is to respond as quickly as possible to end-user requests. Some use strategies to distribute requests fairly among the nodes in the cluster using the kube-proxy component. This does not guarantee a significant reduction in latency given that they do not consider some node hardware resources during the load-balancing process [14]. In this paper, to address the latency problem in an Edge computing-based IoT network using Kubernetes for container orchestration, we first introduce a solution to deploy applications on edge nodes depending on the resources required by these applications to run smoothly on the cluster nodes. Secondly, we propose an approach to manage load-balancing and distribute the requests among the nodes of the cluster while considering hardware resources available on cluster nodes. Thus, the contribution of this paper is summarized as follows

This work is based on the research supported by the National Research Foundation of South Africa (Grant Numbers: 141918)

- We show how the default kube-scheduler can be extended to evaluate and provide a score to applications and deploy them on edge nodes in an efficient way while considering the type of nodes and the available resources.
- We explore the complexities of container orchestration at the edge using Kubernetes, emphasizing the importance of accommodating cluster heterogeneity and implementing request distribution mechanisms for better load-balancing.

The remainder of this paper is organized as follows: Section II reviews the related works. Section III presents our approach to addressing the latency problem, while in Section IV we present the simulation results and discuss the performance of the proposed approach. Finally, Section V concludes the paper and discusses the perspectives of our proposal.

II. Literature Review

This section aims to present the works that are related to our study. We start by presenting kubernetes in Section II-A, then, the state of the art on works that use Kubernetes as orchestrators to reduce the latency in Section II-B.

A. What is on Kubernetes?

Kubernetes is a prominent open-source platform designed to streamline the deployment, management, and scaling of containerized applications. A Kubernetes cluster comprises of master nodes (or control planes) and worker nodes. Applications are run in units called pods, that serve as Kubernetes, the smallest execution units [16]. The control plane manages the worker nodes and the pods within the cluster. In production environments, the control plane operates across multiple computers, and a cluster generally encompasses multiple nodes, ensuring fault tolerance and high availability. Fig. 1 shows the kubernetes cluster architecture.

The control plane components are responsible for making overarching decisions regarding the cluster, such as scheduling, identifying and reacting to cluster events, such as initiating a new pod when necessary. These components have the flexibility to operate on any machine within the cluster. However, to streamline operations, setup scripts commonly initiate all control plane components on a single machine, and also avoid executing user containers on a machine. The control pane has four main components: (1) **kube-apiserver** is a key component also known as API server for the Kubernetes control plane that serves as an interface for the Kubernetes API and acts as its primary gateway. (2) **etcd** is a database within which Kubernetes information like metadata, current state and desired states are stored. (3) **kube-scheduler** is the control plane component that monitors created pods without assigned nodes and selects suitable nodes for their execution. Finally, (4) **kube-controller-manager** is the control plane component responsible for executing controller processes.

The worker nodes are responsible for the applications (pods) that are running in the cluster. They mainly have three components that run on every worker node, with the main role of maintaining the running pods and providing the Kubernetes runtime environment. (1) **kubelet** is a node-level agent that helps to ensure the operational status of containers within pods. (2) **kube-proxy** operates as a network proxy and serves as a fundamental component of the Kubernetes service framework. Finally, (3) **Container runtime** is an essential element that enables Kubernetes to efficiently run containers.

In a Kubernetes cluster, relying on the IP address of an application pod for access can be challenging due to the pod's IP changing upon restart. To ensure application reachability, a layer of abstraction known as a service is employed to expose groups of application pods to clients. Each service is assigned a virtual IP address, called ClusterIP, which remains unchanged unless recreated, thus guaranteeing application reachability within the cluster. However, ClusterIP is only accessible within the cluster. To enable access from outside the cluster, services can be configured with NodePort or LoadBalancer. With NodePort, a static port is opened on each node, allowing access through nodeIP:port from inside or outside the cluster. LoadBalancer leverages the cloud provider's load-balancing mechanisms to externally expose the service. Traffic accessing an application is routed to its pods and monitored by the service, with routing decisions based on kube-proxy modes such as userspace, iptables, and IPVS.

By default, kube-proxy operates in userspace mode, utilizing a round-robin algorithm for traffic distribution. Alternatively, iptables mode randomly selects a pod for traffic handling. For large-scale applications, IPVS offers more efficient routing algorithms including round-robin, least connection, destination hashing, source hashing, shortest expected delay, and never enqueued. Another important component in Kubernetes is the horizontal pod autoscaler (HPA), which is in charge of creating automatically new pods of an application if the existing pods are overloaded, or deleting some of them if the application is not being used.

B. Overview of Works using Kubernetes as Orchestrator

Many recent studies have focused on the integration of Kubernetes for load-balancing management in edge computing-based IoT networks. Although Kubernetes has some limitations in edge computing-based IoT networks, the works presented in study [4, 19] show that it remains a promising candidate for managing load balancing and improving latency and throughput. Cilic et al. [5] found that Kubernetes and its distributions hold promise in facilitating efficient scheduling across edge network resources. Nevertheless, several challenges remain to be addressed to optimize these tools for the dynamic and distributed execution environment inherent in edge computing. In research [10], Han et al. introduced Kais, a learning-based scheduling framework for such edge-cloud systems to improve the long-term throughput rate of request processing. Unfortunately, in their work, some nodes have a high workload, while others are under-utilised. Goethals et al. [8] proposed FLEDGE, A container orchestrator compatible with kubernetes, leveraging Virtual Kubelets, primarily targeting container orchestration on resource-constrained edge devices. They did not integrate the fact that edge nodes are most of the time heterogeneous, and in their implementation, they also did not consider the distribution of user requests among the nodes, which does not allow them to achieve better results in terms of latency and bandwidth. Phuc et al. [16] proposed an extension of the default HPA whose role is to offer the

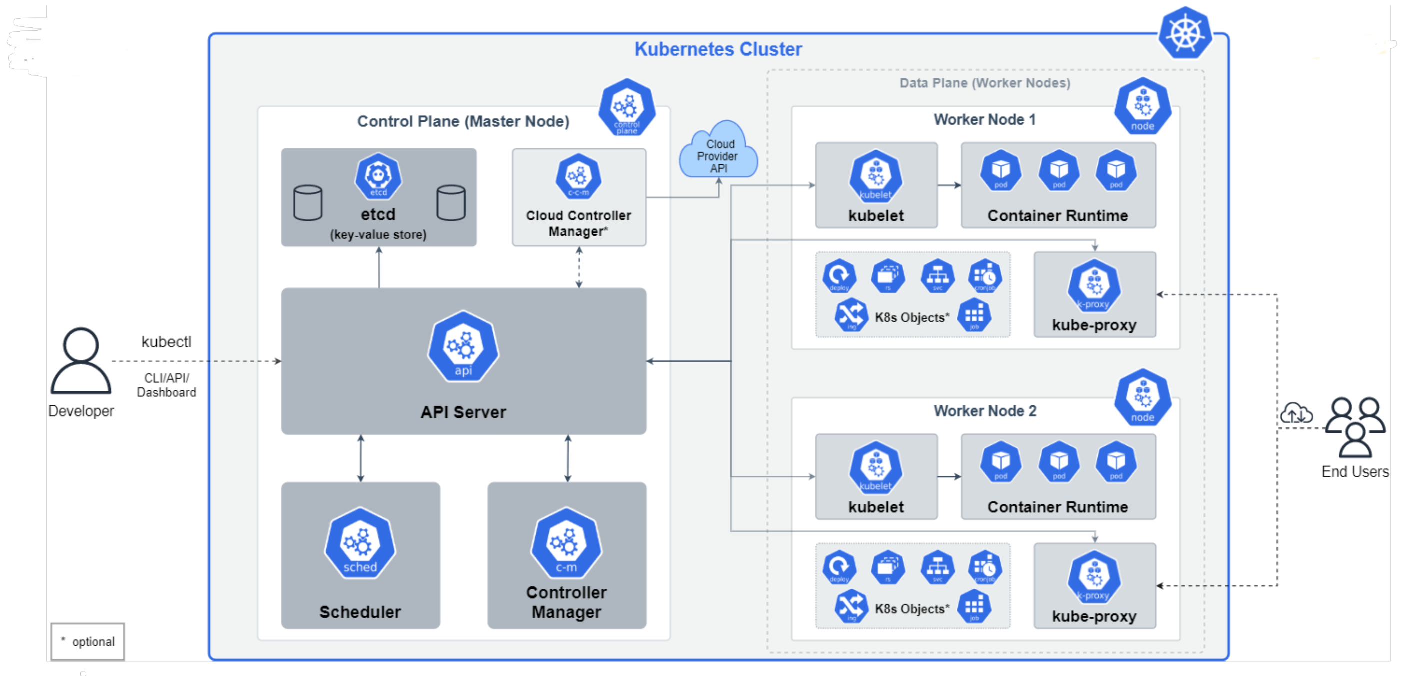


Fig. 1. Kubernetes cluster architecture [2].

resource autoscaling capability. However, the HPA allocates pods to worker nodes without considering the resource demand imbalances in edge computing environments. Authors in [16] considered upscaling and downscaling actions based on network traffic information from nodes to enhance the load-balancing of IoT services in the edge computing infrastructure. Their solution increased throughput and response time between edge nodes, but it doesn't guarantee that a pod that required a lot of resources will be executed on a worker node having the requirements to run that kind of application; therefore, if a particular pod is running on the wrong node, it can increase energy consumption and end-users requests can take too much time to be executed, resulting in high latency. In study [13], the authors studied KubeEdge, which is an open-source platform based on Kubernetes, that aims to orchestrate containerized IoT application services in IoT edge computing environments. In study [6], authors proposed a multi-application hierarchical autoscaling framework tailored for Kubernetes edge clusters. A mechanism based on applications nominates optimal deployment choices through workload prediction and various criteria to ensure application performance while minimizing infrastructure provider costs. This strategy achieves significant improvement in the average allocated resources and energy consumption but it doesn't operate during the first deployment of the application on the servers. Nguyen et al. in study [14], proposed a scheme named Resource Adaptive Proxy (RAP) that considers metrics like latency between worker nodes, Central Processing Unit (CPU) and Random Access Memory (RAM) on worker nodes in order to efficiently distribute the end users requests among the worker nodes. Unfortunately, RAP algorithm does not monitor other cluster resources such as graphics processing units and storage. Rac et al. in STUDY [18] introduce a methodology aimed at reducing the operational costs of applications across the edge-to-cloud computing continuum. Authors in study [17] proposed EdgeOptimizer, a solution that serves as an online testbed for verifying kubernetes-based algorithms, offering detailed configuration options to facilitate cluster management. Unfortunately, their solution does not evaluate the hardware resources consumed by the applications running on their most complex use cases in order to efficiently schedule their deployments in Kubernetes clusters in an edge computing environment.

So far, several works that aim to integrate Kubernetes distributions for container orchestration in edge computing-based IoT networks have been presented. The purpose of doing that is to reduce the latency by managing the load-balancing in edge computing-based IoT networks when considering all the requirements of the edge computing environment. Unfortunately, despite the effectiveness of these solutions, there are still some limitations that need to be addressed while considering Kubernetes for container orchestration in edge computing-based IoT networks.

Edge nodes are heterogeneous and some of them can run faster, some resource-constrained applications than others, and it is therefore important to schedule application deployments by considering the best nodes on which they will efficiently run. Secondly, since Kubernetes and recent works do not integrate network-related metrics and resources like Graphical Processing Unit (GPU) and storage for load-balancing properly. It is important to enhance them and propose solutions to achieve better latency and throughput. Finally, default Ingress-Controller and Kube-proxy components distribute evenly end-users requests among the pods without considering the available hardware resources on the worker nodes. In this work, we address these issues and propose a solution that schedules deployment of containerized applications on worker nodes efficiently, and secondly manages load-balancing by distributing requests among the worker nodes while considering hardware metrics and effective locations of the pods on the worker nodes.

III. CONTRIBUTION

This section presents our contribution for enhancing latency between end-users and edge layer in edge computing-based IoT networks deployed in smart cities using Kubernetes for container orchestration. The strategy we are proposing consists, on the one hand, extension of the default kube-scheduler installed on the master, to enable it to deploy pods while taking into account the minimum hardware resources required to run those pods and the types of machines in the cluster having these hardware resources. On the other hand, enhancing the default Kubernetes load-balancing algorithms and integrating other metrics with CPU and RAM to manage the workload on the edge layer nodes.

A. Assumptions and Notation

The following assumptions are done for this work:

- There are several edge nodes geographically dispersed in a smart city, and thousands of applications are running on them;
- All applications are containerised using docker, and Kubernetes is used for container orchestration;
- The master node has powerful resources and can launch virtual machines for a sandbox environment, which is a dynamic area where developers can play with codes without fear of disrupting the larger system and where they can test their solutions;
- The worker nodes in the cluster are heterogeneous;
- It is possible to evaluate the hardware resources consumed by a running container and assign a score to that container depending on the resources it consumes;
- It is also possible to evaluate a machine and assign it a score according to its resources;

In Table I, we have a list of notations that will be used to describe our solution

B. Kube-scheduler Awareness

We propose an extension to the default kube-scheduler installed on the master node. In Section II-A, we have described the behaviour of the default kube-scheduler and at the end of Section II, we highlighted some of the limitations of this component. To address these limitations, we propose the integration of a robot installed on the master node, that can evaluate the hardware resources consumed by a running container and decide on what kind of nodes it can be deployed. So, to put our proposed solution into practice, developers will have to expose test endpoints for the most resource-intensive use cases in their applications before they are deployed on the cluster. Thus, before the deployment of containers on the cluster, the kube-scheduler performs the following operations:

- When the kube-scheduler receives a new container for deployment purposes on the cluster, it first deploys it in the sandbox environment (virtual machines running on the master node);
- Once the said application is deployed in the sandbox environment, the robot starts and simultaneously tests all the endpoints exposed by the container. The idea is to overload this application as much as possible and monitor the resources it requires when it is running at full speed;
- A score S is given to a container based on the resources it needs when running at maximum speed :
  - If $S \geq T_{appHGPU}$, then the kube-scheduler will consider only nodes having high GPU and high CPU capacity for the deployment of the new container;
  - If $T_{appLGPU} \leq S \leq T_{appHGPU}$, then the kube-scheduler will only consider nodes having low GPU and high CPU capacity for the new container deployment;
  - If $T_{appHCPU} \leq S \leq T_{appLGPU}$, then the kube-scheduler will only consider nodes having acceptable CPU and high GPU capacity for the new container deployment;
  - If $T_{appLCPU} \leq S \leq T_{appHCPU}$, then the kube-scheduler will only consider nodes having acceptable CPU and low GPU capacity for the new container deployment;
- After the score S is assigned to an application, its deployment is faster and easily done because only some nodes will be considered depending on that score;
- Otherwise, all the nodes likely to host a pod are overloaded, the new pod will be deployed on the worker in the higher level.

Since thousands of applications are running at the same time on edge nodes in a smart city, it is important not to over-use or under-use the nodes by choosing which application to run on what kind of node, so that this node will execute user requests with the highest possible speed compared to the other nodes on the network. The parameters that we used to evaluate the score of an application are RAM, CPU, GPU, network bandwidth between nodes's components and storage (HDD or SDD). Each parameter is assigned a weight W: then the weight for the amount of RAM used by an application is W_{RAM}, for the amount of CPU used is W_{CPU}, for storage memory is W_{Stor}, for the GPU is W_{GPU} and for bandwidth is W_{BW}. Also, we define N_i as the amount of hardware resource i consumed by a running application: so, let N_{RAM} be the amount of RAM used by a running application, N_{CPU} the amount of CPU used by the same running application, N_{Stor} the amount of data stored by that application during its execution, N_{GPU} the amount of GPU used and N_{BW} be the amount of bandwidth used by that application. Since the measurement units of each parameter differ, their values have to be normalized. Thus, the score S of a running application is given by:

$$S = \frac{W_{RAM} \times N_{RAM}}{max_{RAM}} + \frac{W_{CPU} \times N_{CPU}}{max_{CPU}} + \frac{W_{Stor} \times N_{Stor}}{max_{Stor}} + \frac{W_{GPU} \times N_{GPU}}{max_{GPU}} + \frac{W_{BW} \times N_{BW}}{max_{BW}} \quad (1)$$

where max_{RAM} is the highest value of RAM memory among all the nodes in the cluster, max_{CPU} is the highest value of CPU among all the nodes in the cluster, max_{Stor} is the highest

TABLE I. NOTATIONS

Notation	Explanation
CPU	Central Processing Unit
GPU	Graphical processing unit
RAM	Random Access Memory
N_i	Amount of hardware resource i consumed by a running application
$T_{appHGPU}$	Threshold value for an application to be launched on nodes having high GPU capacities and high CPU capacities
$T_{appLGPU}$	Threshold value for an application to be launched on nodes having low GPU capacities and high CPU capacities
$T_{appLCPU}$	Threshold value of an application to be launched on nodes having acceptable CPU capacities and low GPU capacities
$T_{appHCPU}$	Threshold value of an application to be launched on nodes having acceptable CPU capacities and high GPU capacities

value of storage memory among all the nodes in the cluster, max_{GPU} is the highest value of GPU among all the nodes in the cluster and max_{BW} is the highest value bandwidth among all the nodes in the cluster.

Algorithm III.1 shows the different stages executed by the Kube-Scheduler before new containers deployment on worker nodes.

Algorithm III.1: Kube-scheduler awareness

Input: App: a containerized application to deploy on the cluster

1 **Begin**
2 Reception of the container (App) to deploy on the cluster ;
3 Deploy container App in the sandbox environment on Master ;
4 Evaluate the score S of container App using equation 1 ;
5 **if** $S \geq T_{appHGPU}$ **then**
6 only consider nodes having high GPU and high CPU capacity for the deployment of App;
7 **if** $T_{appLGPU} \leq S \leq T_{appHGPU}$ **then**
8 only consider nodes having low GPU and high CPU capacity for the deployment of App ;
9 **if** $T_{appHCPU} \leq S \leq T_{appLGPU}$ **then**
10 only consider nodes having acceptable CPU and high GPU capacity for the deployment of App ;
11 **if** $T_{appLCPU} \leq S \leq T_{appHCPU}$ **then**
12 only consider nodes having acceptable CPU and low GPU capacity for the deployment of App ;
13 Deploy container App on the selected list of nodes ;

C. Load-balancing Awareness

In smart cities, end users send thousands of requests to the edge layer of an edge computing-based IoT network. Default load-balancing algorithms implemented on kube-proxy are userspace, iptable and ipvs, which are not suitable in the edge computing environment. In this section, we propose to use RAP protocol [14] where we integrate other metrics like storage and GPU in the nodes evaluation process considering that the number of nodes for the redirection of requests is reduced since custom applications are running on custom nodes. When a request arrives at a node, it performs the following operations:

- If the current node is running the target pod and has sufficient hardware resources, it executes locally the request and sends the response to the user;
- Otherwise, if it is not running the target pod, it sends the later to the node having the best score [14] and that is running the target pod. The redirection is done faster than in [14] because only a few nodes will be considered for the redirection.

So in this approach, user requests are performed mainly local on the node that receives the request, reducing the risk of latency during request redirection. In addition, in the case of redirection, the selection algorithm runs faster because the list of nodes to be considered is reduced since only some nodes run particular applications. In Fig. 2 the default kube-scheduler and HPA deploy applications on worker nodes without considering the minimum requirements of that application. For example, the best nodes that can run App1 are worker 3 and worker 4 but since they are already running an instance of App1, the next instance of App1 is deployed on worker 1 which will not process requests faster on App1 because of its hardware resources. Also, the default userspace algorithm implemented on kube-proxy component for load-balancing distributes requests evenly among the worker nodes. Thus, when worker 4 receives request number 6 to be executed by App1, it forwards it to worker 1 since App1 on worker 1 is only performing request number 3. So in addition to the delay between worker nodes 1 and 4 (7ms), you need to add the time required by worker 1 to perform requests on App1 which is not negligible.

In Fig. 3, pods are deployed on the best nodes that can perform user requests faster. For example, App1 is deployed on worker nodes 3 and 4. Also, user requests are executed locally most of the time, and are only redirected if the pod receiving the request is overloaded or if the node receiving the request is not running the target pod; for example, request number 4 on worker 4 to App2 (in orange) is redirected on worker 1. Therefore, as shown in Fig. 3, the HPA can delete the second instance of App2 on worker 1, and the two instances of App3 on worker 2 since they are not used; this will contribute to free resources on these nodes.

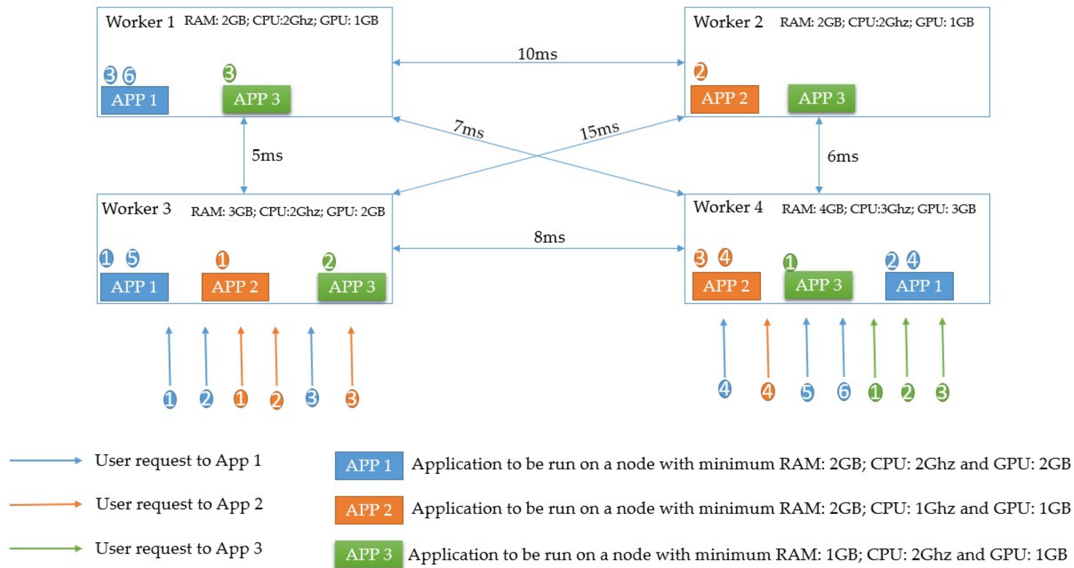


Fig. 2. Default kube-scheduler and userspace algorithm.

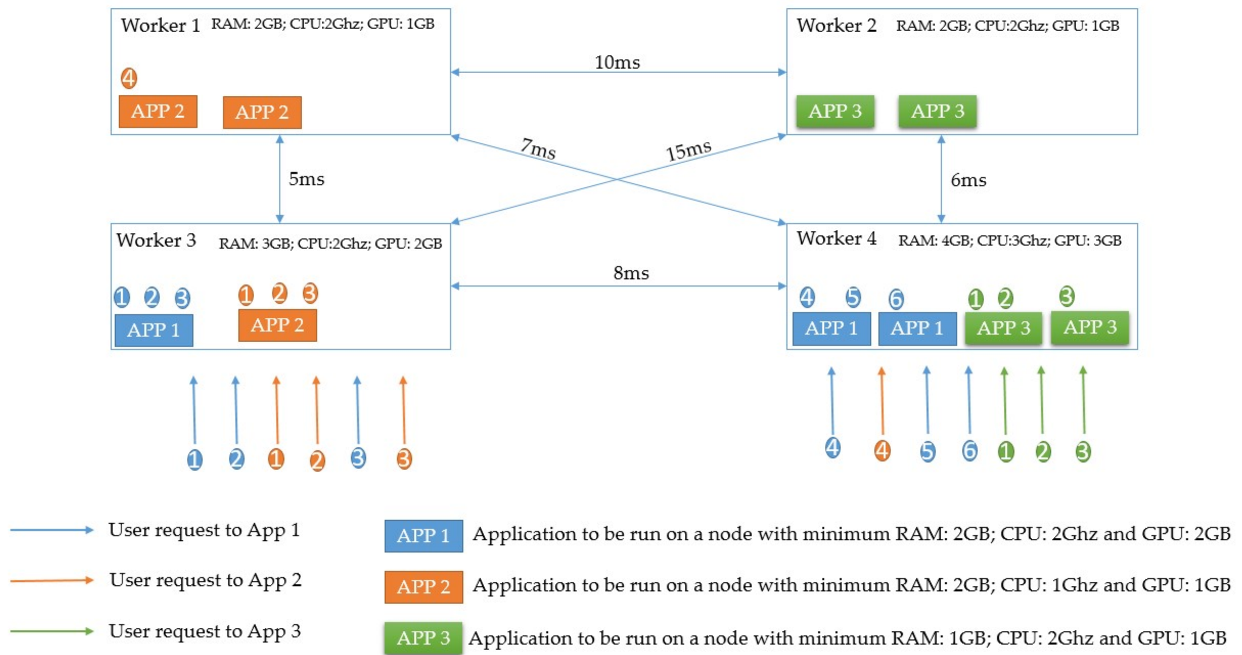


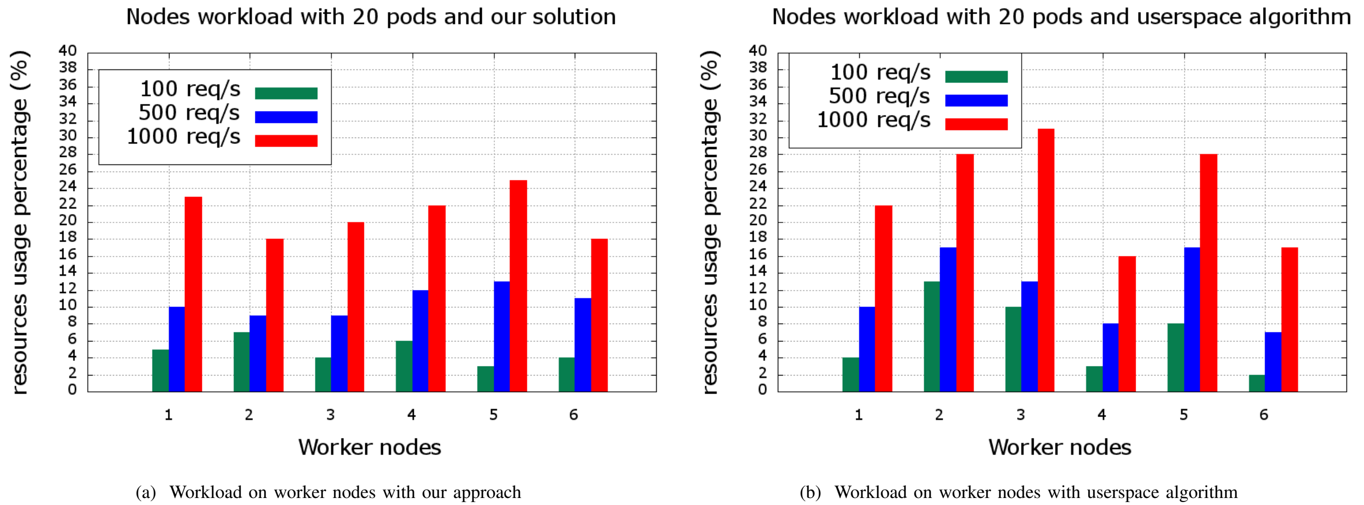
Fig. 3. Our extended kube-scheduler and load-balancing algorithm.

IV. Simulations and Results

In this section, we present the results of our simulations.We installed microk8s V1.26 which is a lightweight distribution of Kubernetes on a cluster with one master node and 6 worker nodes. The characteristics of each node are given in Table II. Also, requests to our Kubernetes cluster have been simulated with the Apache HTTP Server Benchmarking (AB) tool, which is a utility that tests the performance of a server. It has been designed to give an idea of the level of performance of an installation. In particular, it allows one to determine the number of requests an installation can process per second.

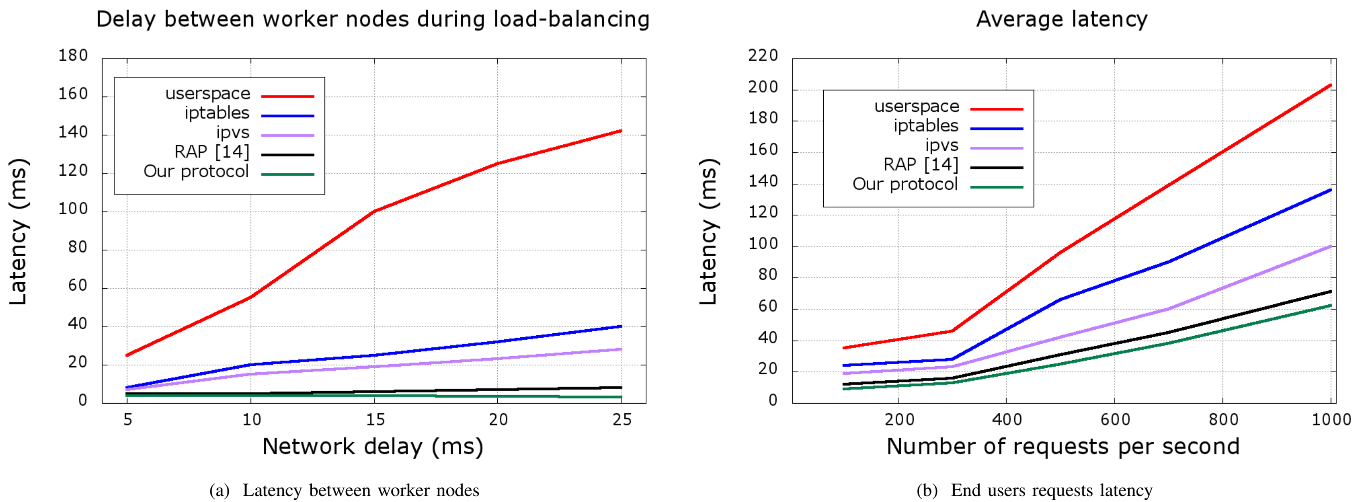
A. Workload on Worker Nodes

Fig. 4 shows the evaluation of the workload on each worker node when there are 100, 500 and 1000 requests per second (req/s) arriving at the cluster with 20 pods. Fig. 4a shows the results when nodes implement our load-balancing algorithm. All the pods are distributed among the worker nodes while considering their resources. The workload is well distributed among the nodes and there are no cases where some nodes are over-used while others are under-used. Fig 4a shows that when 500 req/s are arriving on the cluster, worker 1 uses 10% of its resources, worker 2 uses 9% of its resources, worker 3



(a) Workload on worker nodes with our approach (b) Workload on worker nodes with userspace algorithm

Fig. 4. Evaluation of workload on worker nodes.



(a) Latency between worker nodes (b) End users requests latency

Fig. 5. Latency evaluation.

TABLE II. Nodes Characteristics

Node	Hard drive type	Number of CPU	disc storage (GB)	RAM (GB)	CPU (Ghz)	GPU
Master	SSD	5	250	8	3.5	4
Worker 1	SSD	2	120	3	2.5	2
Worker 2	SSD	3	150	2	1.8	3
Worker 3	SSD	2	100	3	2.8	1
Worker 4	SSD	4	180	2	3.0	2
Worker 5	SSD	2	150	4	2.0	2
Worker 6	SSD	3	100	2	3.1	2

uses 9%, worker 4 uses 12%, worker 5 uses 13% and worker 6 uses 11% of its resources. However, Fig. 4b shows the results when the default userspace algorithm is implemented. It shows that, when 500 req/s are arriving on the cluster, worker 1 uses 10% of its resources, worker 2 uses 17% of its resources, worker 3 uses 13%, worker 4 uses 8%, worker 5 uses 17% and worker 6 uses 7%. Fig. 4b shows that, even if the requests are evenly distributed among the worker nodes, some pods are not deployed on the appropriate worker nodes; and therefore, these nodes are overused. These evaluations have been made while executing the most resource-intensive use cases of each pod. That is, with the use of our proposed solution, pods are deployed on the most suitable nodes in the cluster so that even if they are in demand, they can perform operations and deliver responses faster.

Results shown in Fig. 4 are supported by those presented in Table III, which shows the comparison between our protocol

TABLE III. Evaluation of Standard Deviation

Protocol	100 req/s	500 req/s	1000 req/s
Our protocol	1.37	1.49	2.59
userspace algorithm	3.97	4	5.73

and the default userspace algorithm in terms of the standard deviation of the workload on worker nodes. The standard deviation with our protocol is 1.37 when there are 100 req/s, while the one of the userspace algorithms is 3.97. When there are 500 req/s, the standard deviation with our protocol is 1.49 while the one of userspace is 4; and when there are 1000 req/s, the standard deviation with our protocol is 2.59 while the one of userspace is 5.73. These results show that the workload on worker nodes when using our protocol does not vary a lot compared to the cases with userspace.

B. Evaluation of Delay Between Worker Nodes During Load-balancing and Its Impact on Latency

Fig. 5a shows the evolution of the latency when the network delay between two worker nodes increases. Since in edge computing, worker nodes are geographically dispersed, the latency between worker nodes plays a key role in load-balancing management. Our protocol was compared to the RAP protocol [14] and the default routing decisions implemented in kube-proxy such as userspace, iptables and ipvs. Fig. 5a shows that userspace is the worst because of its round-robin algorithm and its performance varies from 25ms to 140ms; iptable which varies from 8ms to 40ms is better than userspace because while performing its random algorithm it often selects the closest node to transfer requests; ipvs which is tailored for large scale applications perform better than userspace and iptable. RAP protocol [14] is better than all the default routing decisions implemented in kube-proxy because it executes as many as possible requests locally on the node that received the request, and forwards them only if the local node has no capacity; moreover, the forwarding of a request takes into consideration the hardware resource on the other nodes and the delay between them. Our protocol is the best because our routing algorithm is not only based on the one proposed by [14], but also takes less time because the workload is reduced, thanks to the scheduler distributed pods while considering the scores and the nodes on which they can be deployed. Thus for load-balancing purposes, only a few worker nodes are considered in the routing algorithm.

We also evaluated the average latency of our solution when the cluster receives 100 to 1000 requests per second and compared the obtained results with userspace, iptables, ipvs and RAP [14] as presented in Fig. 5b. While latency with userspace varies from 35ms to 203ms, it varies from 24ms to 136ms in iptables, from 19ms to 100ms with ipvs, from 12ms to 71 ms in RAP, and finally from 9ms to 62ms in our solution. Our solution is the best because of two main reasons, the pods are distributed on suitable nodes which perform user requests faster and secondly, it is based on RAP protocol [14] for load-balancing algorithm. Fig. 4a shows that the workload is well distributed among the nodes even when several pods are running, and Fig. 5a shows that even if the delay between worker nodes increases, the latency for load-balancing management between those nodes is still weak. So, the fusion of these results helps to show that the network latency of our solution is the best.

V. Conclusion

This paper presents our approach to enhance latency in an edge computing-based IoT network with Kubernetes as the container orchestrator. This approach extends the default kube-scheduler component deploys pods on the best nodes in the cluster, and then distributes requests among the worker nodes to manage load-balancing. The results show that this approach is promising and better than other approaches in terms of latency. In future work, we plan to examine throughput and a politic for moving containers from one node to another when necessary to increase latency. We also plan to reduce latency by deploying pods according to the geographical regions in which they are used effectively, using machine learning techniques.

References

[1] Hemant Kumar Apat, Rashmiranjan Nayak, and Bibhudatta Sahoo. A comprehensive review on internet of things application placement in fog computing environment. *Internet of Things*, 23:100866, 2023.

[2] The Kubernetes Authors. Kubernetes cluster architecture. https://kubernetes.io/docs/concepts/architecture/, 2024-04-22. 2023-10-23.

[3] Sebastian Böhm and Guido Wirtz. Profiling lightweight container platforms: Microk8s and k3s in comparison to kubernetes. In *ZEUS*, pages 65–73, 2021.

[4] Sebastian Böhm and Guido Wirtz. Cloud-edge orchestration for smart cities: A review of kubernetes-based orchestration architectures. *EAI Endorsed Transactions on Smart Cities*, 6(18), 2022.

[5] Ivan Cilic, Petar Krivic, Ivana Podnar Zarko, and Mario Kusek. Performance evaluation of container orchestration tools in edge computing environments. *Sensors*, 23(8), 2023.

[6] Ioannis Dimolitsas, Dimitrios Spatharakis, Dimitrios Dechouniotis, Anastasios Zafeiropoulos, and Symeon Papavassiliou. Multi-application hierarchical autoscaling for kubernetes edge clusters. In *2023 IEEE International Conference on Smart Computing (SMARTCOMP)*, pages 291–296, 2023.

[7] Raphael Eidenbenz, Yvonne-Anne Pignolet, and Alain Ryser. Latency-aware industrial fog application orchestration with kubernetes. In *2020 Fifth International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 164–171, 2020.

[8] Tom Goethals, Filip De Turck, and Bruno Volckaert. Extending kubernetes clusters to low-resource edge devices using virtual kubelets. *IEEE Transactions on Cloud Computing*, 10(4):2623–2636, 2022.

[9] Tom Goethals, Bruno Volckaert, and Filip De Turck. Adaptive fog service placement for real-time topology changes in kubernetes clusters. In *CLOSER*, pages 161–170, 2020.

[10] Yiwen Han, Shihao Shen, Xiaofei Wang, Shiqiang Wang, and Victor C.M. Leung. Tailored learning-based scheduling for kubernetes-oriented edge-cloud system. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–10, 2021.

[11] Rathinaraja Jeyaraj, Anandkumar Balasubramaniam, Ajay Kumara M.A., Nadra Guizani, and Anand Paul. Resource management in cloud and cloud-influenced technologies for internet of things applications. *ACM Comput. Surv.*, 55(12), mar 2023.

[12] Paridhika Kayal. Kubernetes in fog computing: Feasibility demonstration, limitations and improvement scope : Invited paper. In *2020 IEEE 6th World Forum on Internet of Things (WF-IoT)*, pages 1–6, 2020.

[13] Seong-Hyun Kim and Taehong Kim. Local scheduling in kubeedge-based edge computing environment. *Sensors*, 23(3), 2023.

[14] Quang-Minh Nguyen, Linh-An Phan, and Taehong Kim. Load-balancing of kubernetes-based edge computing infrastructure using resource adaptive proxy. *Sensors*, 22(8), 2022.

[15] Juan Marcelo Parra-Ullauri, Hari Madhukumar, Adrian-Cristian Nicolaescu, Xunzheng Zhang, Anderson Bravalheri, Rasheed Hussain, Xenofon Vasilakos, Reza Nejabati, and Dimitra Simeonidou. kubeflower: A privacy-preserving framework for kubernetes-based federated learning in cloud–edge environments. *Future Generation Computer Systems*, 157:558–572, 2024.

[16] Le Hoang Phuc, Linh-An Phan, and Taehong Kim. Traffic-aware horizontal pod autoscaler in kubernetes-based edge computing infrastructure. *IEEE Access*, 10:18966–18977, 2022.

[17] Yufei Qiao, Shihao Shen, Cheng Zhang, Wenyu Wang, Tie Qiu, and Xiaofei Wang. Edgeoptimizer: A programmable containerized scheduler of time-critical tasks in kubernetes-based edge-cloud clusters. *Future Generation Computer Systems*, 156:221–230, 2024.

[18] Samuel Rac and Mats Brorsson. Cost-effective scheduling for kubernetes in the edge-to-cloud continuum. In *2023 IEEE International Conference on Cloud Engineering (IC2E)*, pages 153–160, 2023.

[19] Rafael Vaño, Ignacio Lacalle, Piotr Sowiński, Raúl S-Julián, and Carlos E. Palau. Cloud-native workload orchestration at the edge: A deployment review and future directions. *Sensors*, 23(4), 2023.

[20] Cecil Wöbker, Andreas Seitz, Harald Mueller, and Bernd Bruegge. Fogernetes: Deployment and management of fog computing applications. In *NOMS 2018 - 2018 IEEE/IFIP Network Operations and Management Symposium*, pages 1–7, 2018.