Geolocate: A geolocation-aware scheduling system for Edge Computing

João Vilaça, João Paulo, Ricardo Vilaça
HASLab - High-Assurance Software Lab, INESC TEC & U. Minho, Portugal.
Email: {joao.p.vilaca,joao.t.paulo,ricardo.p.vilaca}@inesctec.pt

***Abstract*—Edge computing has emerged as an important paradigm that moves the computation and data storage of distributed services closer to users. While virtualization technologies, such as containers, have eased the task of running distributed services in heterogeneous edge hardware, these are still lacking adequate scheduling and orchestration algorithms that can indeed place computation near data producers.**

We present Geolocate, a generic scheduler for workload orchestration and distribution in containerized edge deployments. As the main novelty, the proposed solution takes into account both the available computational resources and the geographical location of nodes when deploying service components in these.

A preliminary experimental evaluation of our prototype shows that Geolocate scheduling time is on par with other schedulers such as the KubeEdge default scheduler. Moreover, Geolocate shows average gains, even on single isolated requests, of about 62% in the response times of scheduled services.

I. Introduction

The Edge Computing paradigm aims at leveraging the computational and storage capabilities of IoT devices (also referred as Edge nodes), while resorting to Cloud Computing services for more demanding processing tasks that cannot be done at commodity devices. However, deploying distributed services across Edge and Cloud nodes raises new challenges that must be addressed. Namely, the choice of what nodes run each service component may be critical for ensuring an efficient service for users [1], [2]. For example, if two critical components, that must frequently exchange data, are placed in different geographic locations, the whole performance of the service will be affected.

Virtualization and orchestration technologies such as Kubernetes [3], widely used in industry, are excellent for managing distributed services running at heterogeneous Cloud nodes [4]. Kubernetes allows to distribute and manage various workloads across a wide variety of nodes but is built for cloud infrastructures, thus not being prepared to accommodate other processing units, such as Edge devices.

To bridge this gap, KubeEdge was created, aiming to extend Cloud capabilities to the Edge [5]. Based on Kubernetes, KubeEdge implements add-ons on the original cluster, networking, node management, and data communication. However, KubeEdge provides simple scheduling algorithms that only take into account the available resources of nodes when deploying service components at these. When Cloud and Edge nodes are running in close geographic locations, the previous scheduling decisions are sufficient. However, when these nodes are far away from each other and must frequently exchange data, the service's latency is significantly affected and, consequently, the experience of users.

For example, we can consider the case of a real-time road data processing service, with sensors that produce large amounts of information. On one hand, it is important that the processing workloads are close to the data sources, which will decrease latencies and the quantity of data sent over the network. On the other hand, it is possible that the generated data may fall under specific data protection and privacy laws, such as the General Data Protection Regulation 2016/67 (GDPR), which limits the transfer of personal data outside the European Union and European Economic Area.

Therefore, the main goal of this paper, for these geographically dispersed environments, is to explore, design, and implement new orchestration and distribution systems for hybrid Cloud and Edge environments, based on geographic location, service demand, business objectives, laws, and regulations.

Throughout this work, we aim to achieve substantial improvements in scalability and quality of service levels by taking advantage of Edge's computational resources. In more detail, the proposed solution must be able to handle various heterogeneous software and hardware environments and reliably ensure its performance requirements. In particular, the protocol must be able to establish processing units on Cloud or Edge nodes, according to the nature of the computation, the type and geographical location of the data.

To address the previous goals, this paper proposes Geolocate, a generic scheduler for workload orchestration and distribution across heterogeneous and geographically distant nodes. In more detail, it provides the following contributions:

- An implementation of a scheduling and placement algorithm based on nodes' geographic location and resource availability.
- A fully functional prototype, integrating Geolocale with KubeEdge, a Edge computing orchestration platform based on Kubernetes.
- A preliminary experimental evaluation of our prototype in a real deployment and comparing different scheduling approaches.

The results show that, under normal cluster conditions, where data-producing workloads are relatively stable, GeoLocate performs well in data processing times, service response times and increases network performance, reducing bandwidth usage and consequently increasing applications throughput. By reducing latency between data-producing and data-processing services, Geolocate is able to decrease overall service response times up to 62%.

II. Related Work

Few studies have addressed the resource allocation and scheduling problem in hybrid Cloud and Edge environments.

Zenith [1] proposes a utility-aware resource allocation solution for Edge Computing with a decoupled model where the management of nodes is independent of the service providers. Dyme [2] is a dynamic microservice scheduling system for mobile edge computing that is able to minimize the total network delay and network price. While both enable the optimization of resource usage and minimize latencies across nodes, these do not take into account the nodes' geographic location in their orchestration decisions.

HYDRA [6] is a decentralized and distributed orchestrator for containerized microservice applications. While it can manage heterogeneous resources across geographical locations and enables the location-aware deployment of microservice applications via containerization, it is a completely new solution implemented from the ground up. Thus, it does not leverage the additional features (e.g., pod abstraction, support for different storage drivers and backends, volume management, maintainability, service discovery, load-balancing, and existing user base) of mature solutions such as Kubernetes.

Indeed, several frameworks extend Kubernetes cloud capabilities to the edge. Among these, some stand out, such as MicroK8s [7], a simple low-sized Kubernetes package that allows users to add edge nodes to a cluster and guarantees high availability through a consensus algorithm. Akri [8] is a Kubernetes interface for Edge. Although it dynamically discovers nodes and provides some management tools, it leaves many management tasks (e.g., nodes and devices configuration, discovery protocol settings and launch of broker pods to handle device information) to the user.

Kubernetes Cluster Federation [9] is an orchestrator of orchestrators, as it is composed of a series of Kubernetes clusters, and introduces the concept of Cross Cluster Service Discovery, enabling developers to deploy a service that was sharded across a federation of clusters spanning different zones, regions or cloud providers. Despite enabling geographic distribution of workloads, the Kubernetes Cluster Federation is, by definition, a multi-cluster, multi-cloud system that cannot operate in an edge computing paradigm since it assumes an underlying high-performance network in each cluster.

Another alternative is KubeEdge [5]. It is a fully transparent abstraction for the Kubernetes API, which allows the management of the cluster with edge nodes with kubectl, the Kubernetes command-line tool. It ensures fault-tolerant and highly available communication between all nodes and provides lightweight agents for the edge.

Given that none of the previous Kubernetes-based solutions provides geo-location scheduling algorithms, the work proposed in this paper builds on top of KubeEdge to provide such a solution. We chose this technology because it is efficient, easy to maintain and endorsed by the Cloud Native Computing Foundation [10].

III. KubeEdge

KubeEdge extends Kubernetes and allows the management of remote edge nodes and edge applications deployments, without changing the Kubernetes API. It provides edge controllers for node and workload handling, a custom network protocol, and a distributed metadata storage, to support system faults and offline scenarios where edge nodes are not connected to the cloud [5].

As depicted in Figure 1, the EdgeController module is responsible for managing the edge nodes. It extends the Kubernetes default controller with edge capabilities, allowing the API Server to integrate the edge nodes in the cluster. The network connection between the cloud and edge nodes is implemented by EdgeHub and CloudHub. These modules are responsible for assuring a fast and reliable communication interface between the cluster nodes.

In terms of workloads, edge applications and resources are configured and controlled by the Edged module, a lightweight agent that packages all the edge node functionality into one process. This module is also responsible for launching and controlling three other modules composing the system, the EventBus, the DeviceTwin, and the MetaManager. These extra modules manage all external Edge devices and data handling.

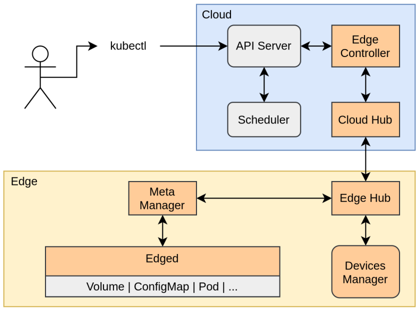

Fig. 1: KubeEdge architecture.

This architecture allows KubeEdge to provide an offline network mode, assure fault tolerance, and high availability while efficiently supporting cross-platform and heterogeneous software and hardware environments. All of these while allowing a simplified development process, with an SDK for systems and resource management, and maintenance.

One of the main components in KubeEdge clusters is the Scheduler, a service responsible for attaching pods, the single most basic instances of a running process in the cluster, to nodes in the system. For each newly created pod or unscheduled pod, the Scheduler selects a node to attach the application and execute it.

However, since one can configure different resource requirements for pods (e.g., to define the minimum CPU and memory necessary for the application to run), existing nodes need to be filtered according to the availability of these resources. Briefly, the Scheduler selects a suitable node in a 2-step operation. In the first, the filtering step, it finds the list of nodes where it is possible to attach the pod. If the list is empty, this pod cannot be deployed. When there is more than one possible node, we enter the scoring step, where the scheduler sorts the remaining nodes by available resources to choose the most appropriate for pod attachment.

However, The KubeEdge Scheduler is still a very naive dynamic resource-provisioning mechanism which only considers nodes' resource utilization, therefore not being very effective [11]. For example, when considering two cluster nodes in two different zones of the planet and a data-producing workload near the first one, the default scheduler instead of choosing the first node to minimize network latency, will ignore this geographic distance aspect and select any of the nodes. Therefore, this paper proposes the first geolocation-aware scheduling system for KubeEdge. With a focus on the geographic location of data-producing workloads, this system can minimize network latencies when deploying consumer pods and improve service response times.

IV. Geolocate

Next, we further describe Geolocate's scheduler design principles, arquitecture and integration with KubeEdge.

A. Design Principles

Accounting for the existence of a data-producing system located in a specific geographic location, the scheduler must be able to, given a corresponding data-processing workload, calculate the best fitting node for its deployment. The selected nodes should minimize the geographic location between data producers and consumers, while improving network latency, data processing delay, and service response time.

As shown in Figure 2, the scheduler must be aware of the location of available computing Edge nodes at different granularities, namely their city, country and continent. When deploying a data processing workload (Figure 2 - 1), the scheduler will allow users to define the location (i.e., city, country, continent) desired for running their processing workloads, for instance, closer to the corresponding data producers.

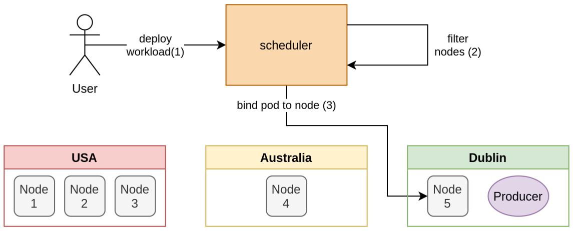

Fig. 2: Geolocator's design principles.

These user-defined locations can be specified as *mandatory*. In this case, if the desired location does not have the necessary computational resources, the scheduler should return an error response, and the workload's deployment should be delayed until enough resources are available. If locations are specified as *preferred* and a node at the desired location is not available, the scheduler will select the closest node with available computing resources. After selecting the best node (Figure 2 - 2), the workload will be deployed (Figure 2 - 3).

The scheduling algorithm should also be generic to allow integration with different orchestration tools other than KubeEdge. Namely, it should include internal mechanisms for abstracting the management of nodes and workloads from the framework where it is being deployed.

B. Scheduler architecture

Following the specification mentioned above, we defined several interfaces and modules. The proposed architecture follows the single-responsibility principle, encapsulating node and workload information in data structures, facilitating the system's extensibility. In Figure 3, we can observe the various components that make up the Geolocate scheduler.

The *scheduler-core* component allows the creation of generic node selection algorithms for workload placement. In this solution, we present an algorithm that takes into account the geographic location of nodes (for example, data-processing and data-producing). It also offers several data structures for indexing cluster information about nodes and workloads. Finally, the *scheduler-core* provides a public interface with all the necessary methods to manage the cluster nodes and execute the scheduling algorithm.

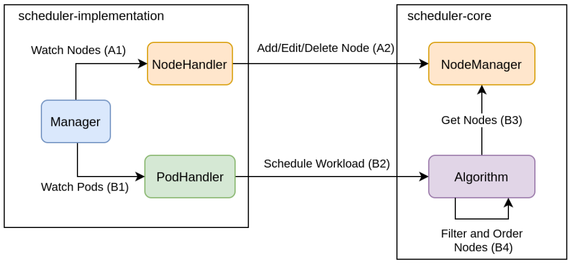

Fig. 3: Scheduler architecture

The *scheduler-implementation* integrates the *scheduler-core*, through its public interface, with the target orchestration tool, in this case, KubeEdge.

The *NodeHandler*, receiving all node changes made to the cluster (Figure 3 - A1), whether adding new ones, editing (including current available resources updates) or removing existing nodes, is responsible for keeping the *Scheduler-core* with up-to-date information through the *NodeManager* (Figure 3 - A2). In turn, the *NodeManager* maintains an internal structure for node management using a map, with geographic locations of the nodes as keys, to optimize the algorithm search for nodes in each of the geographical locations.

The *PodHandler* is responsible for consulting/receiving cluster information about the existence of service workloads associated with the Scheduler that are not yet bound to any node (Figure 3 - B1). When the *PodHandler* gets any pending workload, the placement algorithm is executed (Figure 3 - B2). While no node is found in one of the requested locations, the algorithm fetches from the *NodeManager* the nodes from the next location (Figure 3 - B3), excludes nodes without enough available resources to support this workload (Figure 3 - B4), and if any remains, one is selected.

C. Integration with KubeEdge

KubeEdge has a modular architecture that eases the integration of new components, such as the scheduler proposed in this paper. Namely, Geolocate's prototype is composed by our *Scheduler* mechanism, integrated with the KubeEdge cluster through the creation of a new *Custom Resource* and a corresponding *Controller*.

1) Custom Resource: A *Custom Resource Definition* allows users to create system resource types that do not exist by default in KubeEdge. Our new custom resource, named *EdgeDeployment*, enables users to define the structure of the resource configuration files where are included several fields needed for the workloads scheduling process (e.g., the number of replicas, the mandatory or preferred location, computational requirements).

2) Custom Controller: A *Custom Resource* needs to have an associated Controller that is responsible for ensuring that resource configurations defined by the user are enforced at the KubeEdge cluster. Also, the *Controller* allows updating (e.g., user submits new parameters for preferred locations for a given workload) and deleting existing user configurations being applied at the cluster.

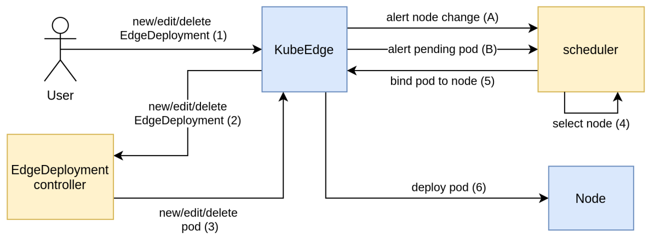

Fig. 4: Scheduler integration with KubeEdge

3) Scheduler: The *Scheduler* component is responsible for calculating and selecting the cluster node where the user's pod should be placed and inform KubeEdge. The entire process of deploying and managing the pod on the node is the responsibility of other KubeEdge components (Figure 4 - 6).

In terms of implementation, the *Scheduler* subscribes to the KubeEdge's *Node Informer* to receive alerts about all changes to nodes in the cluster (Figure 4 - A), allowing it to stay up to date on what nodes exist, what their configurations are, and what available resources do these have.

Whenever a new node is added to the cluster, or its available computational resources (e.g., CPU, RAM) are updated, the *Scheduler* reads the total resource capacity of the node and the configurations of the pods running on it. From the pods' information about their resources needs, the *Scheduler* estimates the total used resources on the node at that time. The node geographical location is extracted from the node labels, manually configured by the system administrator, but future work may include the dynamic calculation of node location in this step.

Secondly, the *Scheduler* also subscribes to the *Pod Informer* to be notified of new pods that are not bound to any cluster node and need to be scheduled (Figure 4 - B). When there is a new pod that needs to be scheduled, the algorithm fetches the location configuration from the pod specification and iterates all nodes trying to get any in the selected location (i.e., given city, country or continent) (Figure 4 - 4).

If the system finds a suitable node, the algorithm finishes and the *Scheduler* instructs KubeEdge to bind the pod to that node (Figure 4 - 5). Otherwise, there are two hypotheses. If the location was specified by the user as *mandatory* the *Scheduler* throws an error, and the pod remains unbounded until a suitable node is available or the EdgeDeployment resource is deleted. If the location was specified as *preferred*, the system will lookup up nodes in a broader area, as close as possible to the desired one. In more detail, if a node is not available at the requested city, then the algorithm searches for a free node at the city's country. If no nodes are available at that country, then the algorithm checks for an available node at the corresponding continent. Note that, for a given workload, a user can specify several cities as viable deployment options. In this case, the previous algorithm is similar but it searches first for nodes at the cities, then at all the countries of specified cities and, finally at the corresponding continents. Since, the location is just a preference, if a suitable node could not be found at this point, a random available node in the cluster is selected. Again, if no nodes are available the *Scheduler* throws an error, and the workload's deployment is delayed until a node is available or the EdgeDeployment resource is deleted.

V. Preliminary Evaluation

In this preliminary evaluation stage, we aim to verify that the use of scheduling algorithms, taking into account the geographical location of the data to be processed, results in substantial improvements in application performance and quality of service.

A. Workloads and collected metrics

The workload used in the experiments is based on a modified KubeEdge service example, 'Data Analytics with Apache Beam' [12], an analytics service to get data from the MQTT broker in-stream format and apply rules on incoming data in real-time.

In more detail, we set up a data-producing application (e.g., running at an Edge node) producing messages at a rate of 4 messages per second and submitting them to an MQTT broker, deployed locally at the same node. Then, a data-processing application, deployed in another node (e.g, Cloud server) fetches messages from the broker and processes them, with a fixed processing delay set to 250ms for each message.

For this simple workload, we assume that the data-processing application is responsible for actively requesting data from the broker. Additionally, it fetches one message at a time and only requests a new one after processing the previous one. Although this is a simple example, it shows the advantages of having a location aware deployment strategy, as discussed next.

We modified the application by creating configuration environment variables that allowed us to control workload parameters such as the time between message creation, the number of messages to create, and the message processing delays. Also, we measured the time it takes between the production of a given message (at the data-producing application) to being completely processed (at the data-processing application).

B. Experimental environment

1) Cluster setup: For all tests, we launched a KubeEdge cluster using 6 commodity servers. Each node had an i3 CPU (4 cores at 3.7 GHz), 8 GB of RAM and a 128 GB SSD disk. Hosts were connected over a shared Gigabit Ethernet network.

For the cluster deployment, we used Kubespray, a tool composed of several Ansible playbooks to automate the process. First, we initialized the two cloud nodes, one master and one worker. The master node runs both the KubeEdge system management pods and the Geolocate scheduler. The worker node will run all cloud-side workloads from the experiments.

Afterward, since there were no tools to automate the KubeEdge initialization process and edge nodes addition to the cluster, we developed our playbook for this task. The playbook first installs the KubeEdge binaries at the master node and starts the *EdgeController* process. Then four servers are configured and added to the cluster as edge nodes. These nodes will run all edge-side workloads for the experiments.

2) Latency across regions: Since our scheduler proposes an efficient geographical distribution of data processing workloads according to the location of corresponding data producers, it is important to ensure that our cluster correctly simulates a global distribution of nodes. As it was not possible to conveniently arrange and test our system with nodes effectively distributed over wide geographic regions, we locally simulated the latency between them as described in Table I.

City	Austin	Dublin	Sydney	Tokyo
Austin	-	-	-	-
Dublin	107	-	-	-
Sydney	191	274	-	-
Tokyo	137	240	160	-

TABLE I: Global latency statistics in milliseconds extracted from https://wondernetwork.com/pings.

To apply this latency to the cluster, we used Chaos Mesh, a tool that features fault injection methods for complex systems on Kubernetes. Using the NetworkChaos Experiment, we considered each of the cluster Edge nodes to be in a city mentioned in Table I. We then configured a *Network Delay action* between them and every other Edge node with the desired latency, causing the expected delays in message sending between pods deployed on those nodes.

C. Results

Now, we discuss the different conducted experiments and analyze the corresponding results.

1) Scheduling overhead: As the first experiment, we measured the execution time of the node selection (scheduling) algorithm for the default KubeEdge scheduler and for Geolocate. In this experiment, each scheduler calculates and assigns a data-processing workload to a node. We executed this test 20 times per scheduler. The average results show that both schedulers have similar average execution time around 300μs. However, Geolocate presents a higher standard deviation, almost 174μs, than the default scheduler, 104μs.

2) Location-aware scheduling gains: To evaluate the use of our new location-aware algorithm, we analyzed the performance gains derived from placing data production and processing workloads in close and far away regions. For every test, we deployed a data-processing workload in the Cloud server. Then, for each region, we deployed a data-producing workload and measured the latency of exchanging messages. Each experiment was repeated 5 times.

Figure 5, shows the variation of the average response time (time to complete the processing task for a single message) of our workload when correlated with different network communication latencies between data producer and processing (consumer) nodes. The points depicted at the figure refer to the latencies at Table I, while the line extrapolates the response time for other latency configurations.

When the latency between the data production and processing systems is set to 50ms, the average processing response time, for a single message, is around 400ms. When the observed latency is 250ms, the average system response time increases to 1000ms.

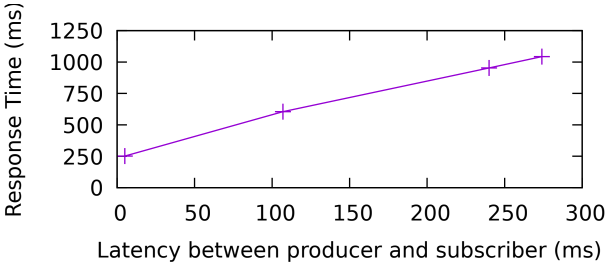

Fig. 5: Latency impact on the underlying system

The latency between the two systems will always have a multiplicative effect on the service's response time. In this case, we verify a proportional increase between the two factors, with a correlation coefficient of 3. In other words, the observed response time equals the message processing time plus three times the latency between producer and subscriber.

A scheduling algorithm that does not addresses geographic location will place, with the same probability, the producer's workload in any node of the cluster, which in average would cause a latency of 157ms and a response time of 700ms. A location-aware scheduler like Geolocate will constantly try to place the workload in the node with lower latency, therefore averaging 5ms, which corresponds to an average service response time of 265ms. This difference represents a 62% gain in service response times.

It is clear the negative influence of high latencies between the systems that produce and consume large amounts of data for different services. Clients using services operating under these conditions will observe high response times because of the high latencies between data-producing and data-processing systems.

3) Geolocate with continuous producer: In terms of placement, let us now consider a cluster with a fixed producing service (e.g., at the Cloud node) generating data messages at a fixed rate (every 250ms) and placed in Dublin. The processing workload fetches those messages and then processes them for 250ms. It can be placed in any of the cities (e.g., at the Edge nodes) described in Table I, which, in these experiments, is achieved with different configurations of the Geolocate's scheduler. In these experiments, we assume that all regions have one cluster node available for the processing task. If this was not the case, the scheduler would choose the available node with the lowest latency.

Figure 6 shows the incremental service response time in processing queued messages according to the pair: (region where data is produced, region where it is processed). For example, in the case (Dublin, Austin), we see that the service processed the 2500th message with a delay of about 8 minutes since the production time. In the 5000th message, this delay was around 16 minutes.

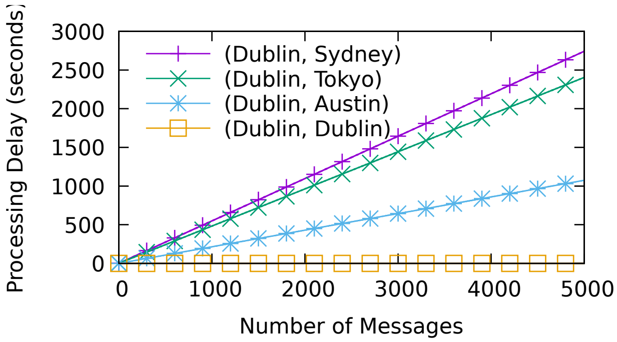

Fig. 6: Service Response Times

Next, we will disregard all the other factors that may condition the placement of workloads besides the geographical location of the data, such as node priorities or balancing their available resources. In these conditions, we know that the default KubeEdge scheduler will place with equal probability the workload on any of the available nodes.

In the best case scenario, with a probability of 25% in this setup, the default KubeEdge scheduler will place the processing workload in the Dublin node, the one with the lowest latency to the data producing service, which keeps the service response time constant at 253 milliseconds.

The worst case scenario also has a probability of 25% in this setup. In this case, the default KubeEdge scheduler would place the processing workload in Sydney. After 2000 messages, the service response time would be around 18 minutes. When considering 5000 messages, this time grows to about 45 minutes.

Using the Geolocate scheduler, the data processing workload is always scheduled in the region defined by users. Therefore, if there are enough available resources and the Dublin location is specified by users, generating in all iterations the pair (Dublin, Dublin) and having a constant response time of 253 milliseconds.

VI. Conclusion

This paper presents Geolocate, a new scheduling solution suitable for data-centric workloads being deployed at geographically distributed Edge environments. The proposed scheduler is integrated with KubeEdge, a state of the art orchestration system that is based on Kubernetes and maintains a similar interface for portability and ease of adoption purposes.

The preliminary evaluation of our prototype shows that Geolocate is able to maintain similar execution times to the KubeEdge default scheduler when calculating scheduling plans. Moreover, it achieves considerable gains in response time due to a better placement of related data-producing and data-processing applications.

The work presented at this paper opens the path to several interesting research questions to be pursued.

1) Additional experiments: The paper shows preliminary experiments with a simple proof-of-concept application. Thus, a detailed evaluation contemplating real applications and different constraints on the number of available computational resources per geographical location needs to be conducted.

2) Applicability: Since KubeEdge is an extension of Kubernetes, the Geolocate's scheduler can be directly applied to other solutions following the same scheduling APIs as Kubernetes. Such integration can be addressed as future work.

3) Automatic and adaptable scheduling: The decision on the location where each workload should run can be improved to be done in an automatic fashion and to be adaptable to workload changes. For this, the scheduling mechanism must be able to gather information about the data being exchanged by applications deployed at Edge and Cloud nodes.

4) Scalability and fault-tolerance: Finally, as future work, it is important to consider the scheduler's scalability and dependability for large-scale deployments.

Acknowledgements

Partially funded by project AIDA – Adaptive, Intelligent and Distributed Assurance Platform (POCI-01-0247-FEDER-045907) co-financed by the European Regional Development Fund (ERDF) through the Operational Program for Competitiveness and Internationalisation (COMPETE 2020) and by the Portuguese Foundation for Science and Technology (FCT) under CMU Portugal.

References

[1] J. Xu, B. Palanisamy, H. Ludwig, and Q. Wang, "Zenith: Utility-aware resource allocation for edge computing," in *2017 IEEE International Conference on Edge Computing (EDGE)*, 2017, pp. 47–54.

[2] A. Samanta and J. Tang, "Dyme: Dynamic microservice scheduling in edge computing enabled iot," *IEEE Internet of Things Journal*, vol. 7, no. 7, pp. 6164–6174, 2020.

[3] "Kubernetes," Jul 2021. [Online]. Available: https://kubernetes.io/

[4] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade," *Queue*, vol. 14, no. 1, p. 70–93, Jan. 2016. [Online]. Available: https://doi.org/10.1145/2898442.2898444

[5] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with kubeedge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, 2018, pp. 373–377.

[6] L. L. Jimenez and O. Schelen, "Hydra: Decentralized location-aware orchestration of containerized applications," *IEEE Transactions on Cloud Computing*, pp. 1–1, 2020.

[7] "Microk8s," Jul 2021. [Online]. Available: https://microk8s.io

[8] "Akri," Jul 2021. [Online]. Available: https://github.com/deislabs/akri

[9] "Kubernetes cluster federation (kubefed)," Jul 2021. [Online]. Available: https://kubernetes.io/blog/2018/12/12/kubernetes-federation-evolution

[10] "Cloud native computing foundation," Jul 2021. [Online]. Available: https://www.cncf.io

[11] C. Chang, S. Yang, E. Yeh, P. Lin, and J. Jeng, "A kubernetes-based monitoring platform for dynamic cloud resource provisioning," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017, pp. 1–6.

[12] "Data analytics with apache beam," Jul 2021. [Online]. Available: https://github.com/kubeedge/examples/blob/master/apache-beam-analysis/README.md