

Sophos: A Framework for Application Orchestration in the Cloud-to-Edge Continuum

Angelo Marchese^a and Orazio Tomarchio^b

Dept. of Electrical Electronic and Computer Engineering, University of Catania, Catania, Italy


Keywords: Cloud-to-Edge Continuum, Containers, Kubernetes Scheduler, Resource-Aware Scheduling, Network-Aware Scheduling, Orchestration.


Abstract: Orchestrating distributed applications on the Cloud-to-Edge continuum is a challenging task, because of the continuously varying node computational resources availability and node-to-node network latency and bandwidth on Edge infrastructure. Although Kubernetes is today the de-facto standard for container orchestration on Cloud data centers, its orchestration and scheduling strategy is not suitable for the management of time critical applications on Edge environments because it does not take into account current infrastructure state during its scheduling decisions. In this work, we present Sophos, a framework that runs on top of the Kubernetes platform in order to implement an effective resource and network-aware microservices scheduling and orchestration strategy. In particular, Sophos extends the Kubernetes control plane with a cluster monitor that monitors the current state of the application execution environment, an application configuration controller that continuously tunes the application configuration based on telemetry data and a custom scheduler that determines the placement for each microservice based on the run time infrastructure and application states. An evaluation of the proposed framework is presented by comparing it with the default Kubernetes orchestration and scheduling strategy.

1 INTRODUCTION

Modern applications such as Internet of Things (IoT), data analytics, video streaming, process control and augmented reality services demand strict quality of service (QoS) requirements, especially in terms of response time and throughput. The orchestration of such applications is a complex problem to deal with (Oleghe, 2021; Calcaterra et al., 2020). Traditional Cloud computing paradigm does not satisfy properly the new computational demands of the applications, characterized by the need for a nearby computation paradigm. Edge computing has emerged as an innovative paradigm able to take advantage of this distributed and close to the end user processing capabilities, thus complementing the Cloud computing ones (Varghese et al., 2021). By combining the Cloud and Edge computing paradigms both the high computational resources of the Cloud and the close to the end user processing capabilities of the Edge are used for the execution of distributed applications.

Kubernetes¹ is a widely adopted orchestration platform that supports the deployment, scheduling and management of containerized applications (Burns et al., 2016). However, Kubernetes has not been designed to work in geo-distributed and heterogeneous clusters such as the aforementioned Cloud-Edge infrastructures (Kayal, 2020; Manaouil and Lebre, 2020). Cloud-Edge infrastructures are dynamic environments, characterized by heterogeneous nodes, in terms of available computational resources, and unstable network connectivity (Khan et al., 2019). In this context, the default Kubernetes scheduling and orchestration strategy presents some limitations because it does not consider the ever changing resource availability on cluster nodes and the node-to-node network latencies (Ahmad et al., 2021). Furthermore Kubernetes does not implement a dynamic application reconfiguration and rescheduling mechanism able to adapt the application deployment to the current infrastructure state and the load and distribution of end user requests. This can lead to degraded application performances and frequent violations on the QoS

^a  <https://orcid.org/0000-0003-2114-3839>

^b  <https://orcid.org/0000-0003-4653-0480>

¹<https://kubernetes.io>

requirements of latency-sensitive applications (Bulej et al., 2020; Sadri et al., 2021).

To deal with those limitations, in this work, based on our previous preliminary works presented in (Marchese and Tomarchio, 2022b; Marchese and Tomarchio, 2022a) we propose Sophos, a framework that runs on top of Kubernetes and extends its control plane to adapt its usage on node clusters distributed in the Cloud-to-Edge continuum. Sophos enhances Kubernetes by implementing a dynamic application orchestration and scheduling strategy able to consider the current infrastructure state when determining a placement for each application microservice and to continuously tune the application configuration and placement based on the ever changing infrastructure and application states and also the current load and distribution of end user requests.

The rest of the paper is organized as follows. In Section 2 we provide some background information about the Kubernetes platform and discuss in more detail some of its limitations that motivate our work. In Section 3 we present the Sophos framework and provide some implementation details of its components, while in Section 4 we provide results of our prototype evaluation on a testbed environment. Section 5 examines some related works and, finally, Section 6 concludes the work.

2 BACKGROUND AND MOTIVATION

Kubernetes is today the de-facto container orchestration platform for the execution of distributed microservices-based applications (Gannon et al., 2017) on node clusters. Kubernetes has been initially thought as a container orchestration platform for Cloud-only environments.

Although new distributions for Edge environments have emerged like KubeEdge, K3s and MicroK8s, Kubernetes is not yet ready to be fully adopted in the Cloud-to-Edge continuum, due to some limitations of its default orchestration and scheduling strategy.

The first limitation is related to the fact that the Kubernetes scheduler does not consider the current state of the infrastructure when taking its decisions. Since the scheduler does not monitor the current CPU and memory utilization on each node, the estimated resource usage may not match its run time value. If resource usage on a node is underestimated, more Pods end up being scheduled on that same node, causing an increase in the shared resource interference between Pods and consequently a decrease in overall applica-

tion performances. Furthermore, the current network latencies between cluster nodes are not considered when evaluating inter-Pod affinities. This means that although an inter-Pod affinity rule is satisfied by placing the respective microservices on the same topology domain, the two microservices are not guaranteed to communicate with low network latencies. High communication latencies between microservices lead to high end-to-end application response times. Considering the run time cluster state and network conditions during Pod scheduling decisions is critical in dynamic environments like the Cloud-to-Edge continuum, where node resource availability and network latencies are unpredictable and highly variable factors.

The second limitation is related to the fact that, although Kubernetes allows application developers to specify the set of application configuration parameters, it does not offer a mechanism to automatically adapt the application configuration based on the current state of the infrastructure and the application itself. All the application configuration is delegated to application developers, who need to predict ahead of time how many resources are required by each microservice and what are the most involved microservices communication channels, in order to determine CPU and memory requirements and inter-Pod affinity weights respectively. However, this is a complex task, considering that microservices resource requirements and communication affinities are dynamic parameters that strongly depend on the run time load and distribution of user requests. Defining Pod resource requirements and inter-Pod affinities before the run time phase can lead to inefficient scheduling decisions and then reduced application performances. Overestimating resource requirements for Pods reduces the probability that these are scheduled on constrained Edge nodes near to end users, while underestimating resource requirements increase Pod density on cluster nodes and then their interference. Errors in estimating inter-Pod affinities can lead to situations where microservices that communicate more are not placed near to each other. Then automatically updating the application configuration at run time is a critical requirement in order to adapt the application deployment based on the current state of the infrastructure and user request patterns.

Finally, another important limitation is related to the fact that Kubernetes does not implement scheduling policies that consider the end user location for the placement of microservices. This is a critical aspect considering that users are typically geo-distributed and the network distance between microservices, especially the frontend ones, and the end user can affect

the application response time.

3 SOPHOS FRAMEWORK

3.1 Overall Design

Considering the limitations described in Section 2, in this work we present Sophos, a framework that extends the Kubernetes platform with a dynamic orchestration and scheduling strategy, in order to adapt its usage to dynamic Cloud-to-Edge continuum environments. The main idea of the proposed approach is that in this context the application orchestration and scheduling task should consider the dynamic state of the infrastructure where the application is executed and also the run time application requirements. To this aim, Sophos monitors the current state of the infrastructure and continuously tunes the application configuration, in terms of microservices resource requirements and inter-Pod affinities, based on the run time application state. Unlike the default Kubernetes platform, in Sophos the configuration of microservices resource requirements and inter-Pod affinities is not anymore delegated to application developers, who should make predictions for these parameters, but it is a dynamic and automated task. Furthermore, the application scheduling strategy is not anymore based on static estimation of node resource availability and network distances between them, but it considers the current node resource usage and the node-to-node network latencies. Finally, in Sophos the application scheduling strategy considers also the end user position. This is a critical aspect to take into account considering the geo-distribution of end users in the Cloud-to-Edge continuum.

Figure 1 shows a general model of the Sophos framework. The current infrastructure and application states are monitored and all the telemetry data are collected by a *metrics server*. For the infrastructure, node resource availability and node-to-node latencies are monitored, while for the application, microservices CPU and memory usage and the traffic amount between them are monitored. Based on the infrastructure telemetry data the *cluster monitor* component determines a *cluster graph* with the set of available resources on each cluster node and the network latencies between them. The *application configuration controller* uses application telemetry data to determine an *application configuration graph* whose nodes represent application microservices with their resource requirements and the edges the communication affinities between them each with a specific affinity weight. The cluster and application graphs are

then used by the *application scheduler* to determine a placement for each application Pod. Further details on the Sophos framework components are provided in the following subsections.

3.2 Cluster Monitor

The cluster monitor component periodically determines the cluster graph with the available CPU and memory resources on each cluster node and the node-to-node network latencies. This component runs as a Kubernetes operator and it is activated by a Kubernetes custom resource, in particular the *Cluster* custom resource. A Cluster resource contains a *spec* property with two sub-properties: *runPeriod* and *nodeSelector*. The *runPeriod* property determines the interval between two consecutive executions of the operator logic. The *nodeSelector* property represents a filter that selects the list of nodes in the cluster that should be monitored by the operator. During each execution of the operator the list of *Node* resources that satisfy the *nodeSelector* condition are fetched from the Kubernetes API server. Then for each node n_i the CPU and memory currently available on it, cpu_i and mem_i respectively, are determined. These values are fetched by the operator from a Prometheus² metrics server, which in turn collects them from node exporters executed on each cluster node. The cpu_i and mem_i parameters are then assigned as values for the labels *available.cpu* and *available.memory* of node n_i .

Then for each pair of nodes n_i and n_j their network cost $nc_{i,j}$ is determined. The $nc_{i,j}$ parameter is an integer value in the range between 1 and 100 and it is proportional to the network latency between nodes n_i and n_j . Network latency metrics are fetched by the operator from the Prometheus metrics server, which in turn collects them from network probe agents executed on each cluster node. These agents are configured to periodically send ICMP traffic to all the other cluster nodes in order to measure the round trip time value. For each node n_i the operator assigns to it a set of labels *network.cost.n_j*, with values equal to those of the corresponding $nc_{i,j}$ parameters. The cluster graph with the updated CPU and memory available resources and the network cost values is then submitted to the Kubernetes API server.

3.3 Application Configuration Controller

The application configuration controller periodically determines the application configuration graph with

²<https://prometheus.io/>

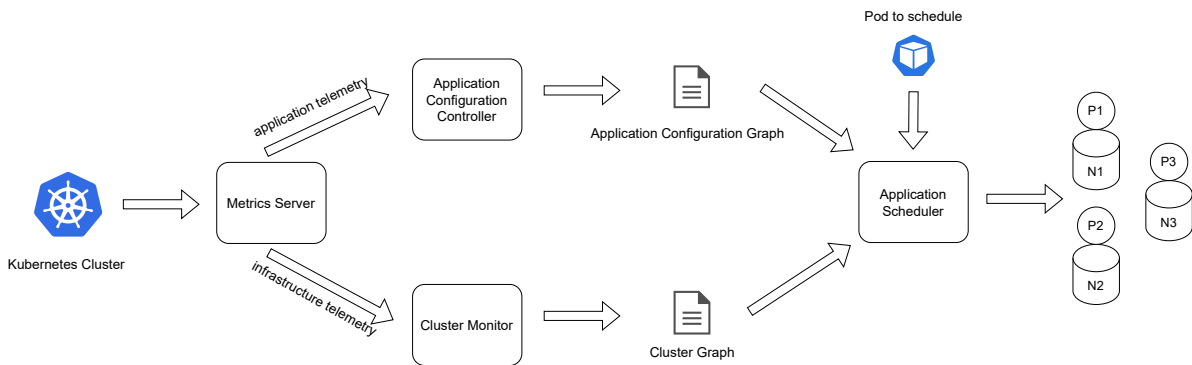


Figure 1: Overall Sophos architecture.

the set of inter-Pod affinity rules between the application microservices. As in the case of the cluster monitor, this component runs as a Kubernetes operator and it is activated by an instance of the *Application* Kubernetes custom resource. An Application resource contains a *spec* property with three sub-properties: *runPeriod*, *name* and *namespace*. The *runPeriod* property determines the interval between two consecutive executions of the operator logic. The name and namespace properties are used to select the set of Deployment resources that compose a specific microservices-based application. The Deployments selected by the Application custom resource are those created in the namespace specified by the namespace property of the custom resource and with a label *app-group* whose value is equal to the value of the name property in the custom resource.

During each execution of the operator, the list of *Deployment* resources selected by the name and namespace properties are fetched from the Kubernetes API server. Then for each Deployment D_i its CPU and memory requirements, cpu_i and mem_i respectively, are determined. These values are equal to the average CPU and memory consumption of all the Pods managed by the Deployment D_i and are fetched by the operator from the Prometheus metrics server, that in turn collects them from *CAdvisor* agents. These agents are executed on each cluster node and monitor current CPU and memory usage for the Pods executed on that node. The cpu_i and mem_i parameters are then assigned as values for the properties *spec.resources.requests.cpu* and *spec.resources.requests.memory* of the Pod template Pt_i .

Then, the operator determines for each Deployment D_i the set of inter-Pod affinity weights $aff_{i,j}$ with all the other Deployments D_j of the application and the set of inter-Pod affinity weights $paff_{i,j}$ with the set of Pods P_j managed by the *inputProxy* DaemonSet Kubernetes resource. This DaemonSet de-

ployes on each cluster node n_i a proxy Pod P_i that intercepts all the user requests arriving at that node and forwards them to the first microservice in the application graph, like for example an API gateway microservice. The $aff_{i,j}$ weight is proportional to the traffic amount exchanged between microservices μ_i and μ_j , while the $paff_{i,j}$ weight is proportional to the traffic amount exchanged between the microservice μ_i and the proxy Pod P_j . Both $aff_{i,j}$ and $paff_{i,j}$ parameters are integer values normalized in the range between 1 and 100. Traffic metrics are fetched by the operator from the Prometheus metrics server, which in turn collects them from Envoy proxy containers injected by the Istio³ platform on each application Pod. By injecting Envoy proxies also on the *inputProxy* Pods the source of user requests and the distribution of user traffic among the different sources can be traced.

Each $aff_{i,j}$ and $paff_{i,j}$ parameter of the Deployment D_i is assigned by the operator as the weight of a corresponding *preferredDuringSchedulingIgnoredDuringExecution* inter-Pod affinity rule in the Pod template Pt_i . The application configuration graph with the set of updated affinity rules is then submitted by the controller to the Kubernetes API server. If at least one affinity rule in the Pod template section of a Deployment has changed with respect to the last iteration, a rolling update process is activated and new Pods with the updated resource configurations are created.

3.4 Application Scheduler

The proposed application scheduler is a custom scheduler, based on the Kubernetes *scheduling framework*, that extends the default Kubernetes scheduler by implementing a set of plugins. In particular, a *ResourceAware* plugin that extends the scoring phase of the Kubernetes scheduler and a *NetworkAware* plugin that extends both the sorting and scoring phases

³<https://istio.io>

are proposed. The node scores calculated by the ResourceAware and NetworkAware plugins are added to the scores of the other scoring plugins of the default Kubernetes scheduler.

The $Score()$ function of The ResourceAware scheduler plugin is invoked during the scoring phase to assign a score to each cluster node when scheduling a Pod p . The function evaluates the Pod p CPU and memory resource requirements, cpu_p and mem_p , as specified by the Pod $spec.resources.requests.cpu$ and $spec.resources.requests.memory$ properties, and for each node n_i the currently available CPU and memory resources, cpu_{n_i} and mem_{n_i} , as specified by the node $available.cpu$ and $available.memory$ labels. The score assigned to the node n_i is given by Equation (1) where α and β parameters are in the range between 0 and 1 and their sum is equal to 1:

$$score(p, n_i) = \alpha \times \frac{cpu_{n_i} - cpu_p}{cpu_p} \times 100 + \beta \times \frac{mem_{n_i} - mem_p}{mem_p} \times 100 \quad (1)$$

The higher the difference between available resources on node n_i and those requested by Pod p , the greater the score assigned to node n_i . By excluding the scoring results of the other scheduler plugins, Pod p is placed on the node with the highest amount of available resources. Unlike the default Kubernetes scheduler that estimates resources availability on a node based on the sum of the requested resources of each Pod running on that node the ResourceAware plugin considers current node resource usage based on run time telemetry data. This allows to reduce the shared resource interference between Pods resulting from incorrect node resource usage estimation and then its impact on application performances.

The $Score()$ function of the NetworkAware scheduler plugin is invoked during the scoring phase to assign a score to each cluster node when scheduling a Pod p . This function evaluates the inter-Pod affinity rules of the Pod p , where in our approach affinity rules are determined by the application configuration controller. Unlike the default Kubernetes scheduler, the NetworkAware plugin does not evaluate the $topologyKey$ parameter in the affinity rules, but it takes into account network cost labels of each cluster node determined by the cluster monitor. Algorithm 1 shows the details of the $Score()$ function.

The algorithm starts by initializing the variable cmc to zero. This variable represents the total cost of communication between the Pod p and all the other Pods (both microservices and proxy Pods) when the Pod p is placed on node n . The algorithm iterates through the list of cluster nodes $cNodes$. For each cluster node cn the $pcmc$ variable value is calculated. This variable represents the cost of communication between the Pod p and all the other Pods $cn.pods$ cur-

rently running on node cmc when the Pod p is placed on node n . For each Pod cnp the $aff_{p,cnp}$ weight of the corresponding affinity rule ($pa_{p,cnp}$ in case cnp is a proxy Pod) is multiplied by the network cost $nc_{n,cn}$ between node n and node cn and added to the $pcmc$ variable. The $pcmc$ variable value is then added to the cmc variable. The final node score is represented by the opposite of the cmc variable value.

Algorithm 1: NetworkAware plugin Score function.

Input: $p, n, cNodes$

Output: $score$

```

1:  $cmc \leftarrow 0$ 
2: for  $cn$  in  $cNodes$  do
3:    $pcmc \leftarrow 0$ 
4:   for  $cnp$  in  $cn.pods$  do
5:      $pcmc \leftarrow pcmc + nc_{n,cn} \times aff_{p,cnp}$ 
6:   end for
7:    $cmc \leftarrow cmc + pcmc$ 
8: end for
9:  $score \leftarrow -cmc$ 

```

The $Score()$ function assigns a score to each cluster node n so that the Pod p is placed on the node, or in a nearby node in terms of network latency, where the Pods with which the Pod p has the greatest communication affinity are executed. For each affinity rule the default Kubernetes scheduler assigns a score different from zero only to the nodes that belong to a topology domain matched by the $topologyKey$ parameter of the rule. The NetworkAware plugin instead scores all the cluster nodes based on the current relative network distance between them. This allows to implement a more fine-grained node scoring approach able to take into account the ever changing network conditions in the cluster instead of using node labels statically assigned before the run time phase.

The $Less()$ function of the NetworkAware plugin is invoked during the sorting phase in order to determine an ordering for the Pod scheduling queue. This function takes as input two Pods resources and returns *true* if the value of the *index* label on the first Pod is lesser than the value of the same label on the second one, otherwise it returns *false*. This way Pods with lower values of the index label are scheduled first. The index label is used to determine a topological sorting of the microservices graph and application developers have to assign a value for this label on the Pod templates of each application Deployment, with lower values given to the frontend microservices. By scheduling frontend microservices first, communication affinities between these microservices and the proxy Pods are evaluated earlier during the scoring phase, resulting in the application placement follow-

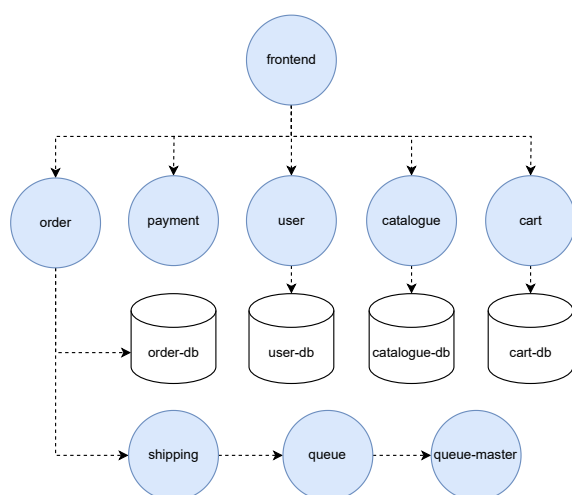


Figure 2: Sock Shop application structure.

ing the end user position. If backend microservices are instead scheduled before the frontend ones, only internal communication affinities between application microservices are evaluated during the scoring phase and this can lead to situations where the frontend microservices are placed far away the end user.

4 EVALUATION

The proposed solution has been validated using the Sock Shop⁴ application executed on a test bed environment. As shown in Figure 2 the application is composed of different microservices, database servers and a message broker. The *frontend* service represents the entry point for external user requests that are served by backend microservices that interact between them by means of network communication. The application can be thought of as composed of different microservice chains, each activated by requests sent to a specific application API.

The test bed environment for the experiments consists of a Rancher Kubernetes Engine (RKE) cluster with one master node and four worker nodes. These nodes are deployed as virtual machines on a Proxmox⁵ physical node and configured with 4GB of RAM and 2 vCPU. In order to simulate a realistic Cloud-to-Edge continuum environment with geodistributed nodes, network latencies between cluster nodes are simulated by using the Linux traffic control (*tc*)⁶ utility. By using this utility network latency delays are configured on virtual network cards of the cluster nodes.

⁴<https://microservices-demo.github.io>

⁵<https://www.proxmox.com>

⁶<https://man7.org/linux/man-pages/man8/tc.8.html>

We evaluate the end-to-end response time of the Sock Shop application when HTTP requests are sent to the frontend service. Requests to the application are sent through the k6⁷ load testing utility. Each experiment consists of 5 trials, during which the k6 tool sends requests to the frontend service for 25 minutes. For each trial, statistics about the end-to-end application response time are measured and are averaged with those of the other trials of the same experiment. The trial interval is partitioned into 5 minutes sub-intervals. During each sub-interval the k6 tool sends requests to a different worker node and in the last sub-interval requests are equally distributed to all the nodes. This way a realistic scenario with variability in the user requests distribution is simulated by dynamically changing the source of traffic at different time intervals. For each experiment we compare both cases when the proposed scheduler and operators of the Sophos framework are deployed on the cluster and when only the default Kubernetes scheduler is present. The α and β parameters of the ResourceAware plugin of the Sophos application scheduler are assigned the same value of 0.5. We consider three different scenarios based on the network latency between cluster nodes: 10ms, 50ms, 100ms.

Figure 3 illustrates the results of the three experiments performed, each for a different scenario, showing the 95th percentile of the application response time as a function of the number of virtual users that send requests to the application. In all the cases, the Sophos framework performs better than the default Kubernetes platform with average improvements of 28%, 43% and 54% in the three scenarios respectively. Furthermore, the improvement is higher for higher node-to-node network latencies. This can be explained by the fact that for low node-to-node latencies, network communication has no significant impact on the application response time. However, when network latencies increase, network communication becomes a bottleneck for application performances and the lack of a network-aware scheduling strategy causes an increase in the application response time.

5 RELATED WORK

In the literature, there is a variety of works that propose to extend the Kubernetes platform in order to adapt its usage to Cloud-Edge environments.

A network-aware scheduler is proposed in (Santos et al., 2019), implemented as an extension of the filtering phase of the default Kubernetes scheduler. The

⁷<https://k6.io>

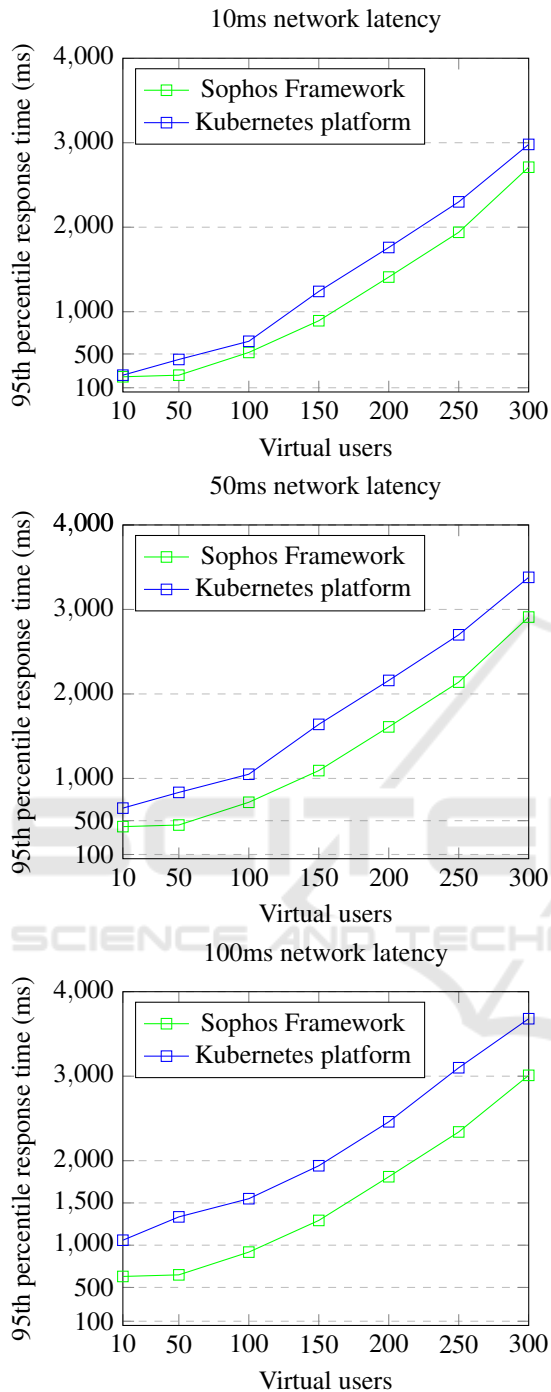


Figure 3: Experiments results.

proposed approach makes use of round-trip time labels, statically assigned to cluster nodes, in order to minimize the network distance of a specific Pod with respect to a target location specified on its configuration file. One problem with this solution relates to the fact that round-trip time labels are statically pre-assigned to cluster nodes, not reflecting the run-time

variability of network latencies.

In (Caminero and Muñoz-Mansilla, 2021) an extension to the Kubernetes default scheduler is proposed that uses information about the status of the network, like bandwidth and round trip time, to optimize batch job scheduling decisions. The scheduler predicts whether an application can be executed within its deadline and rejects applications if their deadlines cannot be met. Although information about current network conditions and historical job execution times is used during scheduling decisions, communication interactions between microservices are not considered in this work.

The authors of (Cao and Sharma, 2021) propose to leverage application-level telemetry information during the lifetime of a distributed application to create service communication graphs that represent the internal communication patterns of all components. The graph-based representations are then used to generate colocation policies of the application workload in such a way that the cross-server internal communication is minimized. However, in this work scheduling decisions are not influenced by the cluster network state.

In (Pusztai et al., 2021) Pogonip, an edge-aware scheduler for Kubernetes, designed for asynchronous microservices is presented. Authors formulate the placement problem as an Integer Linear Programming optimization problem and define a heuristic to quickly find an approximate solution for real-world execution scenarios. The heuristic is implemented as a set of Kubernetes scheduler plugins. Also in this work, there is no Pod rescheduling if network conditions change over time.

In (Wojciechowski et al., 2021) a Kubernetes scheduler extender is proposed that uses application traffic historical information collected by Service Mesh to ensure efficient placement of Service Function Chains (SFCs). During each Pod scheduling, nodes are scored by adding together traffic amounts, averaged over a time period, between the Pod's microservice and its neighbors in the chain of services executed on those nodes. As in our work, historical application traffic is measured, but the proposed scheduler does not take into account current node-to-node latencies, neither communication patterns between microservices.

In (Fu et al., 2021) Nautilus is presented, a runtime system that includes, among its modules, a resource contention aware resource manager and a communication-aware microservice mapper. While the proposed solution migrates application Pod if computational resources utilization is unbalanced among nodes, there is no Pod rescheduling in the

case of degradation on the communication between microservices.

6 CONCLUSIONS

In this work we proposed Sophos, a framework that extends the Kubernetes platform in order to adapt its usage to dynamic Cloud-to-Edge continuum environments. The idea is to add a layer on top of Kubernetes to overcome the limitations of its mainly static application scheduling and orchestration strategy. Orchestrating applications in the Cloud-to-Edge continuum requires a knowledge of the current state of the infrastructure and to continuously tune the configuration and the placement of the application microservices. To this aim in Sophos a cluster monitor operator monitors the network state and the resource availability on cluster nodes, while an application configuration operator dynamically assigns inter-Pod affinities and resource requirements on the application Pods based on application telemetry data. Based on the current infrastructure state and the dynamic application configuration, a custom scheduler determines a placement for application Pods.

As a future work we plan to improve the infrastructure and application monitoring modules with more sophisticated techniques that allow to do predictive analysis on the infrastructure and the application state and to make proactive application reconfiguration and rescheduling actions. To this aim time series analysis and machine learning techniques will be explored in the future.

REFERENCES

- Ahmad, I., AlFailakawi, M. G., AlMutawa, A., and Alsalman, L. (2021). Container scheduling techniques: A survey and assessment. *Journal of King Saud University - Computer and Information Sciences*.
- Bulej, L., Bures, T., Filandr, A., Hnetyinka, P., Hnetynková, I., Pacovsky, J., Sandor, G., and Gerostathopoulos, I. (2020). Managing latency in edge-cloud environment. *CoRR*, abs/2011.11450.
- Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes: Lessons learned from three container-management systems over a decade. *Queue*, 14(1):70–93.
- Calcaterra, D., Di Modica, G., and Tomarchio, O. (2020). Cloud resource orchestration in the multi-cloud landscape: a systematic review of existing frameworks. *Journal of Cloud Computing*, 9(49).
- Caminero, A. C. and Muñoz-Mansilla, R. (2021). Quality of service provision in fog computing: Network-aware scheduling of containers. *Sensors*, 21(12).
- Cao, L. and Sharma, P. (2021). Co-locating containerized workload using service mesh telemetry. In *Proceedings of the 17th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '21, page 168–174, New York, NY, USA. Association for Computing Machinery.
- Fu, K., Zhang, W., Chen, Q., Zeng, D., Peng, X., Zheng, W., and Guo, M. (2021). Qos-aware and resource efficient microservice deployment in cloud-edge continuum. In *IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 932–941.
- Gannon, D., Barga, R., and Sundaresan, N. (2017). Cloud-native applications. *IEEE Cloud Computing*, 4:16–21.
- Kayal, P. (2020). 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.
- Khan, W. Z., Ahmed, E., Hakak, S., Yaqoob, I., and Ahmed, A. (2019). Edge computing: A survey. *Future Generation Computer Systems*, 97:219–235.
- Manaouil, K. and Lebre, A. (2020). Kubernetes and the Edge? Research Report RR-9370, Inria Rennes - Bretagne Atlantique.
- Marchese, A. and Tomarchio, O. (2022a). Extending the kubernetes platform with network-aware scheduling capabilities. In Troya, J., Medjahed, B., Piattini, M., Yao, L., Fernández, P., and Ruiz-Cortés, A., editors, *Service-Oriented Computing*, pages 465–480, Cham. Springer Nature Switzerland.
- Marchese, A. and Tomarchio, O. (2022b). Network-aware container placement in cloud-edge kubernetes clusters. In *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 859–865, Taormina, Italy.
- Oleghe, O. (2021). Container placement and migration in edge computing: Concept and scheduling models. *IEEE Access*, 9:68028–68043.
- Pusztai, T., Rossi, F., and Dustdar, S. (2021). Pogonip: Scheduling asynchronous applications on the edge. In *IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 660–670.
- Sadri, A. A., Rahmani, A. M., Saberikamarposhti, M., and Hosseinzadeh, M. (2021). Fog data management: A vision, challenges, and future directions. *Journal of Network and Computer Applications*, 174:102882.
- Santos, J., Wauters, T., Volckaert, B., and De Turck, F. (2019). Towards network-aware resource provisioning in kubernetes for fog computing applications. In *IEEE Conference on Network Softwarization (NetSoft)*, pages 351–359.
- Varghese, B., de Lara, E., Ding, A., Hong, C., Bonomi, F., Dustdar, S., Harvey, P., Hewkin, P., Shi, W., Thiele, M., and Willis, P. (2021). Revisiting the arguments for edge computing research. *IEEE Internet Computing*, 25(05):36–42.
- Wojciechowski, L., Opasiak, K., Latusek, J., Wereski, M., Morales, V., Kim, T., and Hong, M. (2021). Netmarks: Network metrics-aware kubernetes scheduler powered by service mesh. In *IEEE INFOCOM 2021 - IEEE Conference on Computer Communications*, pages 1–9.