Communication Aware Scheduling of Microservices-based Applications on Kubernetes Clusters

Angelo Marchese[a] and Orazio Tomarchio[b]
Dept. of Electrical Electronic and Computer Engineering, University of Catania, Catania, Italy

Keywords: Container Scheduling, Kubernetes Scheduler, TOSCA, Application Modeling, Network Metrics.

Abstract: Edge computing paradigm has enabled new application categories with low latency requirements. Container technologies are increasingly spreading to provide flexible and scalable services also within these dynamic environments. However, scheduling distributed microservices applications in the Cloud-to-Edge continuum is a challenging problem, considering the instability and limited network connectivity of Edge infrastructure. Existing container orchestration systems, like Kubernetes, allow to ease the deployment and scheduling of distributed applications in Cloud data centers but their scheduling strategy presents some limitations when dealing with latency critical applications, because it does not consider application communication requirements. In this work we propose an extension of the default Kubernetes scheduler that takes into account microservices communication requirements, modeled through the use of the TOSCA language, traffic history and network latency metrics in order to assign node scores when scheduling each application Pod. A qualitative analysis of the proposed scheduler is presented with a use case.

1 INTRODUCTION

The scheduling of container-based microservices applications within node clusters is becoming a challenging problem, especially in arising Fog and Edge computing scenarios where specific quality of service (QoS) requirements exist (Calcaterra et al., 2020; Haghi Kashani et al., 2020; Rodriguez and Buyya, 2019). In the above scenarios, several kind of applications such as process control, augmented reality, smart vehicles and data analysis generate a high amount of data traffic while at the same time require stringent response times. In this regard it becomes necessary to optimize the container scheduling strategies in order to meet this type of requirements (Varshney and Simmhan, 2019; Salaht et al., 2020; Oleghe, 2021; Calcaterra et al., 2021).

Kubernetes[1] is a widely adopted orchestration platform that supports the deployment, scheduling and management of containerized applications. Kubernetes clusters are typically deployed in Cloud data centers with high-performance nodes interconnected through high-bandwidth network links. Although various Kubernetes distributions suitable for Cloud-Edge environments have recently been proposed (Manaouil and Lebre, 2020), its usage in the context of geo-distributed and dynamic environments presents some limitations. One of such limitations relates to its scheduler (Kayal, 2020). While the base Kubernetes scheduling strategy aims to improve resource utilization and balance load on cluster nodes, it is not optimized to deal with the communication requirements of microservices applications and does not evaluate network conditions to take its decisions (Ahmad et al., 2021). Taking into account also communication aspects for decision about application scheduling is essential to improve QoS experimented by end users, especially in the case of latency-sensitive applications (Sadri et al., 2021), (Bulej et al., 2021).

In this work we propose an extension of the base Kubernetes scheduler in the form of a custom scoring plugin that takes into account application communication requirements, traffic history and network latency metrics to assign node scores when scheduling each application Pod. The proposed scheduler consider the application topology graph, the communication patterns and protocols used by its microservices and the amount of traffic exchanged between them in order to determine microservices communication affinity. This information, combined with run-time

[a] https://orcid.org/0000-0003-2114-3839
[b] https://orcid.org/0000-0003-4653-0480
[1] https://kubernetes.io

Marchese, A. and Tomarchio, O.
Communication Aware Scheduling of Microservices-based Applications on Kubernetes Clusters.
DOI: 10.5220/0011049300003200
In *Proceedings of the 12th International Conference on Cloud Computing and Services Science (CLOSER 2022)*, pages 190-198
ISBN: 978-989-758-570-8; ISSN: 2184-5042

node-to-node latencies, is used to optimize the placement of Pods by reducing the network distance between microservices with high communication affinities. We use the TOSCA standard (OASIS, 2020) to model the application topology because of its capabilities to represent in a high-level way the architecture of an application in terms of components and relationships between them. To this purpose we propose a custom relationship types hierarchy in order to model typical communication patterns in a microservice application.

The rest of the paper is organized as follows. In Section 2 we provide some background information about the base Kubernetes scheduler architecture and the TOSCA standard and some related works are also presented. Section 3 discusses the motivations and the design of the proposed scheduler extension, while section 4 provides some implementation details. Section 5 presents a qualitative analysis of the proposed scheduler extension when dealing with the placement of the microservices of a sample messaging application. Finally, Section 6 concludes the work.

2 BACKGROUND AND RELATED WORK

2.1 Kubernetes Scheduler

Kubernetes is a production-grade container orchestration platform which automates the management of cloud-native applications on large-scale computing infrastructures (Burns et al., 2016). Kubernetes orchestrates the placement and execution of application containers within and across node clusters. A Kubernetes cluster consists of a control plane and a set of worker nodes. The control plane is executed by different components that run inside a master node. The worker nodes are responsible for the execution of containerized application workloads. In particular, application workloads consist of *Pods*, which in turn contain one or more containers. A Pod represents the minimal deployment unit in Kubernetes.

Among control plane components, the kube-scheduler[2] is in charge of selecting an optimal cluster node for each Pod to run them on, taking into account Pod requirements and node resources availability. Each Pod scheduling attempt is split into two phases: the scheduling cycle and the binding cycle, which in turn are divided into different sub-phases. During the scheduling cycle a suitable node for the Pod to schedule is selected, while during the binding cycle the scheduling decision is applied to the cluster by reserving the necessary resources and deploying the Pod to the selected node. Each sub-phase of both cycles is implemented by one or more plugins, which in turn can implement one or more sub-phases. The Kubernetes scheduler is meant to be extensible. In particular, each scheduling phase represents an extension point which one or more custom plugins can be registered at. We take advantage of this extensibility property to propose our custom scoring plugin.

[2]https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler

2.2 TOSCA Standard

TOSCA is a standard designed by OASIS that defines a metamodel for specifying both the structure of a Cloud application as well as its life cycle management aspects (OASIS, 2020). The TOSCA language introduces a YAML-based grammar that allows to describe the structure of a Cloud application as a service template, which is composed of a topology template and a set of types needed to build such a template. The topology template defines the topology model of a service as a typed directed graph, whose nodes, called *node templates*, model the application components, and edges, called *relationship templates*, model the relations occurring among such components. Each node template is associated with a node type that defines the properties of the corresponding component. In our work, the use of the TOSCA standard aims to model the topology of a microservice-based application, in particular the relationships between microservices in terms of their communication interactions.

2.3 Related Work

In the literature, there is a variety of works that propose extensions of the default Kubernetes Pod scheduling strategy in order to deal with the communication requirements of modern latency-sensitive and data-intensive applications, especially those executed in Cloud-Edge environments.

In (Rossi et al., 2020) an orchestration tool is presented that extends Kubernetes with adaptive autoscaling and network-aware placement capabilities. The authors propose a two-step control loop, in which a reinforcement learning approach dynamically scales container replicas on the basis of the application response time, and a network-aware scheduling policy allocates containers on geo-distributed computing environments. The scheduling strategy uses a greedy heuristic that takes into account node-to-node latencies to optimize the placement of latency-sensitive applications. The problem with the proposed solution is that it optimizes the placement of each microservice without considering interactions with other microservices.

In (Wojciechowski et al., 2021) a Kubernetes scheduler extender is proposed that uses application traffic historical information collected by Service Mesh to ensure an 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 neighboors 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 (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.

In (Toka, 2021) a Kubernetes edge-scheduler is presented that takes into account the node-to-node network latencies, and the applications' latency requirements in the scheduling decisions about the application components. The scheduler is based on an heuristic algorithm, which optimizes the placement of latency sensitive applications in a geographically distributed infrastructure. A re-scheduler is responsible for container migration in order to improve resource utilization in the infrastructure. In this work, however, microservice-to-microservice communication patterns and exchanged traffic amount are not taken into account during scheduling decisions.

In (Nastic et al., 2021) a scheduling framework is proposed which enables edge sensitive and Service-Level Objectives (SLO) aware scheduling in the Cloud-Edge-IoT Continuum. The proposed scheduler extends the base Kubernetes scheduler and makes scheduling decisions based on a service graph, which models application components and their interactions, and a cluster topology graph, which maintains current cluster and infrastructure-specific states. However, this work also does not consider historical information about traffic amount exchanged between microservices in order to determine their run time communication affinity.

Like the aforementioned research works, in our work we propose an extension of the default Kubernetes scheduler that makes its scheduling decisions based on the type of communication between microservices, the historical traffic amount exchanged between them and the current network state of the cluster. A distinctive contribution of our work, is the application-awareness of the proposed scheduling strategy, aimed to take into account the topology graph of a microservices-based application.

3 SCHEDULER DESIGN

3.1 Motivation

Although Kubernetes is the de-facto container orchestration platform for Cloud environments, its default scheduling strategy presents some limitations when dealing with node clusters dislocated in the Cloud-to-Edge continuum (Kayal, 2020). Edge environments consists of geographically distributed nodes with high heterogeneity in terms of computational resources and network connectivity. Edge infrastructure is more dynamic and unstable than that of Cloud data centers and is characterized by more frequent node failures and network partitions (Khan et al., 2019). Furthermore, recent applications such as Internet of Things (IoT), data analytics, augmented reality and video streaming services demand stronger requirements in terms of response latency and network bandwidth than traditional web applications (Gedeon et al., 2019). All of this poses serious challenges in ensuring these new Quality of Service (QoS) requirements are met (Haghi Kashani et al., 2020), a task that the default Kubernetes scheduler isn't able to handle well for several reasons.

One problem relates to the variety of metrics and node resource types that are considered during each scheduling decision. The scheduler mainly considers CPU and memory limits and requests that are specified in the Pod resource description. However it doesn't consider Pods network bandwidth requirements and its availability on cluster nodes and node-to-node latency, which are critical parameters to schedule communication-intensive and latency sensitive applications. Kubernetes allows to define extended resources for which it is possible to specify requests and limits in the Pod description, but, as for CPU and memory resources, the scheduler doesn't consider their actual utilization in each cluster node when making its scheduling decisions. To evaluate the utilization level of a node resource, the scheduler considers the sum of the values required for that resource by each Pod running on that node. This means that the estimated resource utilization may not match with its run-time value. Considering the run-time cluster state and network conditions during scheduling decisions is critical in dynamic environments like Edge clusters. In addition, the Kubernetes scheduler only evaluates the current view of the system, not taking into account the historical information related to resource utilization, node and network failures and application performances. Another limitation, probably the most important, relates to the application-awareness level of the scheduling decisions. Kubernetes allows to specify resource requirements, Pod-to-node and Pod-to-Pod affinity and anti-affinity rules for Pod scheduling. However the Kubernetes scheduler isn't able to make its placement decisions based on the application topology and microservice-to-microservice communication patterns and traffic history.

In this work we just propose an extension for the Kubernetes scheduler with the aim to overcome some of the aforementioned limitations, in particular those related to network communication between microservices.

3.2 Overall Architecture

The proposed Kubernetes scheduler extension is designed to ensure that network communication aspects are also taken into account during each Pod scheduling decision. In particular, the aim is to ensure that each scheduling decision is also made according to the topology of the application to be deployed, the communication patterns between microservices, the history related to the amount of traffic exchanged between them and the run-time network state, in terms of the communication latencies between cluster nodes. Run-time network conditions and application communication requirements are essential information to optimize container placement decisions in order to reduce end-to-end latencies experimented by end users (Aral et al., 2019).

The basic idea of the proposed scheduler is that, when scheduling a microservice Pod, the type and the amount of traffic exchanged with other microservices and the current node-to-node latencies are taken into account. In a microservices-based application a generic user request consists of a chain of sub-requests and its end-to-end latency is determined by the sum of the latencies of each service call in the chain. This means that in order to obtain lower end-to-end latencies it is not sufficient to minimize the network latency between end users and the front-end components of an application, but it is necessary to consider the whole application topology graph and try to schedule each microservice on the basis of the other microservices placement.

The proposed scheduler extension operates with the aim to minimize the network distance between the microservices that communicate through critical communication channels. Critical communication channels are those channels that use synchronous and blocking communication protocols or those through which a high amount of traffic is exchanged. In general, critical communication channels are those that represent bottlenecks in a chain of requests. Scheduling microservices that communicate through critical channels on the same node or in nodes with a limited end-to-end network latency allows to reduce application response times.

The extension is realized in the form of a custom scoring plugin. This plugin extends the base Kubernetes scheduler scoring phase and, for each Pod to schedule, it assigns a score to each candidate node that passes the filtering phase. The node scoring algorithm of the custom plugin takes as inputs two parameters: the node-to-node communication latency values and the microservice-to-microservice communication affinity values. The former are determined by real time network measurements performed by network probes and collected by a metrics server. The latter are associated to each communication channel between two microservices and are a combination of a static and a dynamic contribution. The static contribution depends on the communication pattern and protocol of the corresponding channel. The more synchronous the communication channel protocol, the higher this value. The dynamic contribution is directly proportional to the average value of the amount of traffic exchanged through the communication channel up to that time. Microservices that communicate through critical channels are those with the higher communication affinity values. Further details of the application topology modeling and node scoring algorithm are provided in the following subsections.

3.3 Application Topology Modeling

One fundamental aspect of the proposed scheduler extension is that it also makes its decisions on the basis of the application topology graph. This graph contains the microservices that make up an application as nodes and the communication channels used to communicate between them as links.

Unlike the base Kubernetes scheduler implementation that mainly considers low level resource requirements, the proposed extension allows to spec-

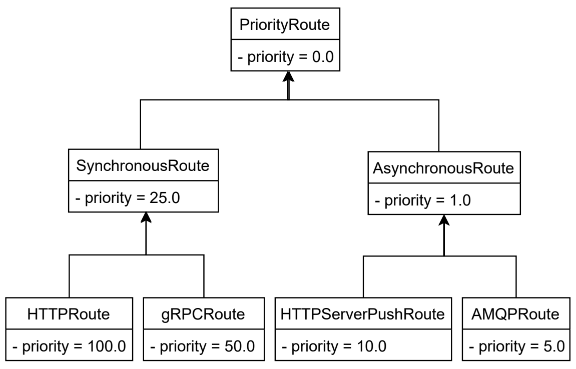

Figure 1: Relationship types hierarchy.

ify higher level requirements that relate with the application composition and its microservice-to-microservice communication patterns. The idea behind our reasoning is that a scheduling strategy based on application architecture and topology information can improve the quality and effectiveness of microservice deployments and allows to realize scheduling strategies customized for the specific application needs. High-level requirements can also relate to the resiliency, availability and reliability of an application. However, in this work we mainly consider those related with the communication aspects, because they are the most critical when dealing with communication-intensive and latency sensitive applications.

A fundamental requirement in the application modeling task is to give the application architect a tool that allows him to model the application topology in a high level and qualitative manner, following an intent modelling approach (Tamburri et al., 2019). This way the application modeler does not have to deal with quantitative parameters and low level details, but he only needs to know the microservices that make up an application and the protocols they use to communicate between them. For this purpose we propose the use of the TOSCA standard for modeling the application topology graph. The application topology graph is modeled as a TOSCA service template where node templates represent microservices and the relationship templates the communication channels between them.

To model the various types of communication channels that characterize typical microservice applications, a custom hierarchy of relationship types is defined, shown in Figure 1. The relationship type *PriorityRoute* is the base type for which the property *priority* is defined. This property represents the priority level associated with the corresponding communication channel and for each type in the hierarchy a default value is assigned. The higher this value, more critical the communication channel is. The relationship types hierarchy is defined in such a way that a higher priority value is associated with the synchronous and heavyweight communication protocols. As an example, relationship types derived from *SynchronousRoute* have a higher priority than those derived from *AsynchronousRoute*. The proposed relationship types hierarchy is only a sample hierarchy that models typical communication patterns in a microservices application. It can be extended with new custom types and the default priority values can be overridden in each relationship template inside a service template.

3.4 Scoring Algorithm

The proposed scheduler extension is realized in the form of a custom scoring plugin. For each Pod to be scheduled, the plugin assigns a score to each candidate node of the cluster that has passed the filtering phase. The scores calculated by the custom plugin are then added to the scores of the other scoring plugins. Algorithm 1 shows the details of the scoring algorithm of the proposed plugin.

Algorithm 1: Node scoring algorithm.

```
Input: $pod, node, channels, latencies, clusterNodes$
Output: $score$
 1: $score \leftarrow 0$
 2: for $cn$ in $clusterNodes$ do
 3:     $pScore \leftarrow 0$
 4:     for $p$ in $cn.pods$ do
 5:         if $areNeighbors(pod, p)$ then
 6:             $sc \leftarrow \alpha \times channels[pod, p].priority$
 7:             $dc \leftarrow \beta \times channels[pod, p].traffic$
 8:             $pScore \leftarrow pScore + sc + dc$
 9:         end if
10:     end for
11:     if $cn == node$ then
12:         $score \leftarrow score + \gamma \times pScore$
13:     else
14:         $\delta \leftarrow 1/latencies[node, cn]$
15:         $score \leftarrow score + \delta \times pScore$
16:     end if
17: end for
```

The algorithm takes as inputs the Pod to be scheduled, the node to be scored, the list of channels in the application graph and the set of nodes in the cluster. The final node score is the weighted sum of partial scores calculated for each cluster node. The weight associated with each partial score depends on whether the corresponding cluster node is the node to be scored or not. If it is, the weight is the pa-

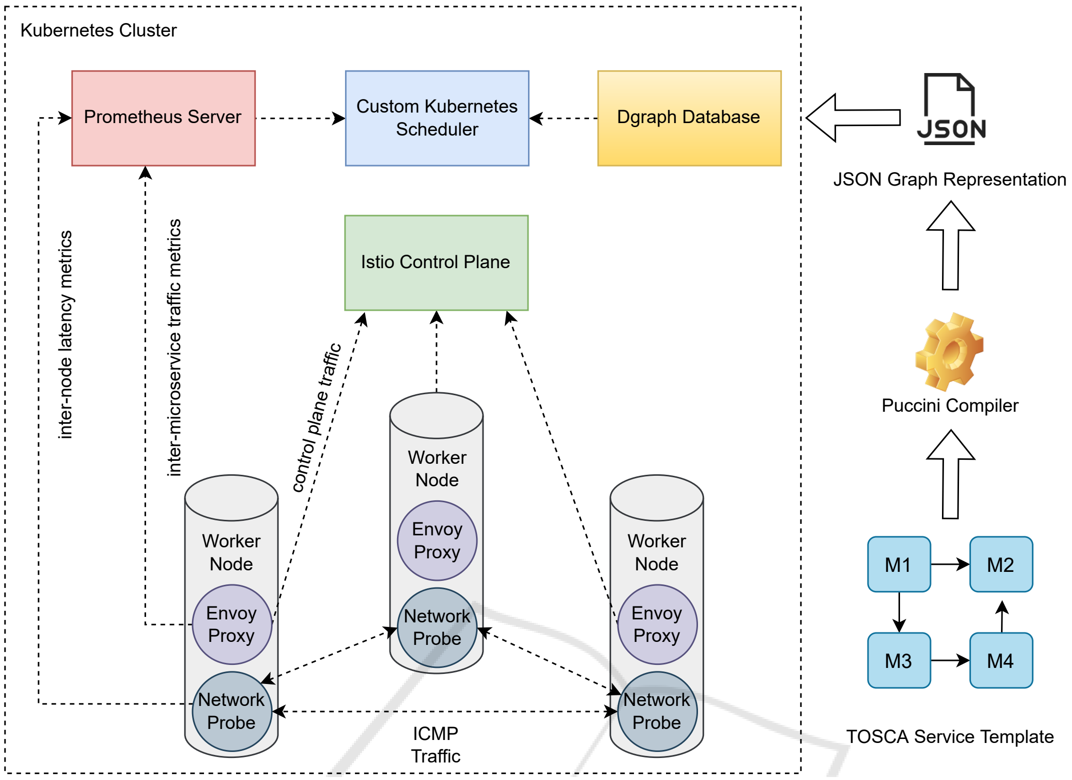

Figure 2: Scheduler extension deployment architecture.

rameter γ. If it is not, the weight is represented by the inverse of the network latency, averaged over a configurable time period, between the corresponding cluster node and the node to be scored. Each partial score is calculated by adding together the communication affinities between the Pods deployed on the corresponding cluster node and the Pod to be scheduled. The communication affinity between two Pods is represented by the sum of a static contribution and a dynamic contribution. The static contribution is the product between a parameter α and the priority value associated with the communication channel between the microservices of the two Pods. The dynamic contribution is the product between a parameter β and the amount of traffic, averaged over a configurable time period, exchanged between the microservices of the two Pods.

The scoring algorithm assigns a score to each node so that the Pod to be scheduled is deployed on the node, or in a nearby node in terms of network latency, on which the microservices with which the Pod has the greatest communication affinity are executed. By tuning parameters α and β, it is possible to modify the contribution on the final node score due to the communication channels priority level and the run-time traffic exchanged between microservices respectively. By giving a higher value to the parameter α, scheduling decisions are mainly influenced by the communication protocols rather than by the amount of traffic exchanged between microservices. On the other hand, by increasing the value of the β parameter, the traffic history takes on greater importance than the type of communication. Similarly, increasing the value of the parameter γ increases the probability that a Pod will be scheduled in the same node where the microservices with which the Pod has the greatest communication affinity are deployed.

4 PROTOTYPE IMPLEMENTATION

Figure 2 shows the deployment architecture of the proposed scheduler extension and the components with which it interacts to make its scheduling decisions. The scheduler extension is executed as a secondary scheduler that is deployed together with the default one. This custom scheduler extends the base Kubernetes scheduler by implementing a scoring plugin written in the Go language. To establish node scores, the plugin uses run-time cluster and application metrics and application topology graph information. Run-time metrics are periodically queried from a Prometheus[3] server by using the Prometheus Go client library[4]. Topology graph information are queried for each Pod to schedule from a Dgraph[5] database by using the Dgraph Go client library[6]. The topology graph of an application is stored in the Dgraph database starting from a JSON representation of the graph. This representation is obtained by compiling the TOSCA service template of the application with the Puccini compiler[7].

The Puccini tool also defines a TOSCA profile for Kubernetes that we extend to define the custom relationship types hierarchy described in Section 3.3. In particular, this profile defines the *Service* node type that we use to model a generic microservice. This node type exposes the capabilities *metadata*, *service* and *deployment*, whose types are *Metadata*, *Service* and *Deployment* respectively. Furthermore, a *route* requirement is defined, whose instances can be associated with *Service* capabilities of other *Service* node templates. The association between *route* requirement instances and *Service* capabilities can be done through *Route* relationship templates. The *Route* relationship type is the base type extended by our custom relationship types hierarchy.

Inter-node latency metrics are collected by network probes deployed in each node of the cluster as Pods of a DaemonSet. Each network probe pings periodically all the nodes in the cluster and export the latency metrics to the Prometheus server. Microservice-to-microservice traffic metrics are collected and exported to the Prometheus server by Envoy proxies executed in each application Pod. Envoy proxies are automatically injected as sidecar containers in each Pod by the Istio framework whose control plane is deployed in the cluster.

5 USE CASE

This section illustrates an example of how the proposed Kubernetes scheduler extension operates when dealing with the placement of the Pods of a typical microservices-based application whose components communicate with each other through both synchronous and asynchronous channels.

The use case shows in a qualitative way how, unlike the default Kubernetes scheduler, the proposed scheduler extension takes into account cluster network conditions and microservices communication

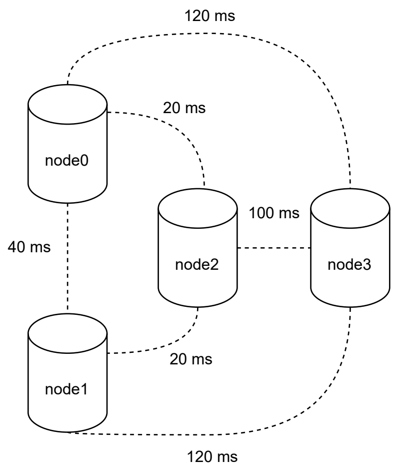

Figure 3: Sample Kubernetes cluster.

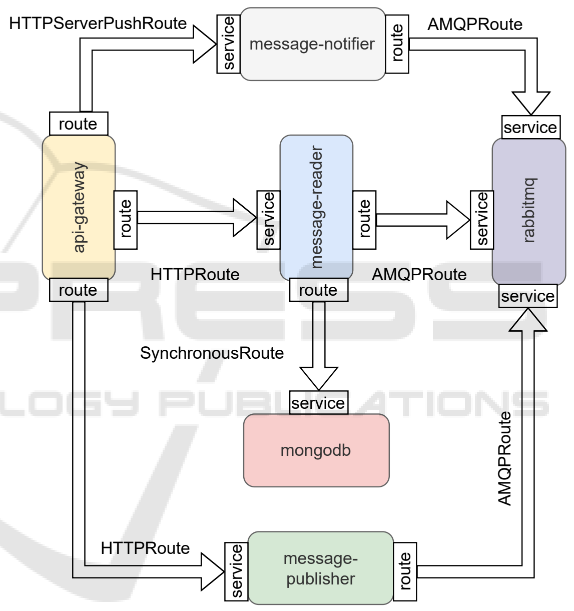

Figure 4: Sample application TOSCA service template.

affinities in order to optimize the placement of application Pods. Considering network state and application communication requirements is critical in the context of Cloud-to-Edge cluster deployments where node-to-node latencies are not negligible.

In this scenario we consider a geographically distributed Kubernetes cluster of four nodes shown in Figure 3. We assume for simplicity that mean node-to-node latencies (shown in the figure) do not change over time. Figure 4 shows the TOSCA service template of a sample messaging application whose components are configured to be scheduled by a secondary scheduler that implements the proposed custom plugin. This application consists of different microservices, modeled as node templates, that commu-

[3]https://prometheus.io
[4]https://github.com/prometheus/client_golang
[5]https://dgraph.io
[6]https://github.com/dgraph-io/dgo
[7]https://puccini.cloud/

nicate with each other through various types of communication channels, modeled as relationship templates. Each node template is an instance of the *Service* node type, defined in the Puccini TOSCA profile, and each relationship template, that associates a *route* requirement with a *service* capability, is an instance of one of the relationship types presented in Section 3.3. The *message-publisher* service receives text messages from users through HTTP requests and publishes them to the *rabbitmq* broker. The *message-reader* service receives messages from the broker and stores them in the *mongodb* database. These messages are then returned to users when the *message-reader* service receives HTTP requests. The *message-notifier* service receives messages from the broker and publishes them in the body of Server Sent Events (SSE) sent through the HTTP/2 Server Push protocol. The *message-publisher*, *message-reader* and *message-notifier* services are exposed to external users through the *api-gateway* service.

For the custom plugin to function correctly it is necessary to deploy first the metrics server, in this case a Prometheus server, the network probes in each node as a DaemonSet and the graph database, in this case a Dgraph database. Then the graph representation of the sample application needs to be stored on the Dgraph database. This representation is obtained by compiling the TOSCA service template of the sample application using the Puccini compiler and its TOSCA profile for Kubernetes extended with our relationship types hierarchy.

For the presented sample scenario we assume that cluster nodes have 4GB of RAM and all microservices Pods requires 1.5GB of RAM, except for the *rabbitmq* service that requests 3GB of RAM. This way a maximum of two application Pods can be deployed on the same cluster node and the *rabbitmq* service Pods can be deployed on nodes where no other Pods are executed. Assuming that the sample application has not been executed yet and the scheduling queue contains Pods for the services *rabbitmq*, *mongodb*, *message-notifier*, *message-reader*, *message-publisher* and *api-gateway* in the specified order, the proposed scheduler plugin assigns node scores in the following way.

For the *rabbitmq* Pod the plugin assigns the minimum score to all nodes because no other dependent Pod is deployed on the cluster. Suppose the *rabbitmq* Pod is scheduled on *node3*, so no other Pods can be scheduled on that node for the assumptions we made earlier. In the same way, for the *mongodb* Pod the same score is assigned to all nodes, so we suppose it is scheduled on *node2*. For the *message-notifier* Pod, the maximum score is assigned to *node2* because this is the closest node to *node3* where the *rabbitmq* service is deployed. This way the remaining Pods can be scheduled only on nodes *node0* and *node1*. When scheduling the *message-reader* Pod, the custom plugin assigns the same scores to *node0* and *node1*, because they have the same network distance to node2 and *node3*, where *mongodb* and *rabbitmq* Pods are deployed respectively. Suppose the *message-reader* is scheduled on *node1*. In the same way, for the *message-publisher* Pod the same score is assigned to both nodes, and for node load balancing reasons this Pod is deployed on *node0*. Finally, for the same reasons the *api-gateway* Pod can be indifferently scheduled on *node0* and *node1*, and we suppose that it is deployed on the latter. 5 shows the resulting placement of the sample application microservices.

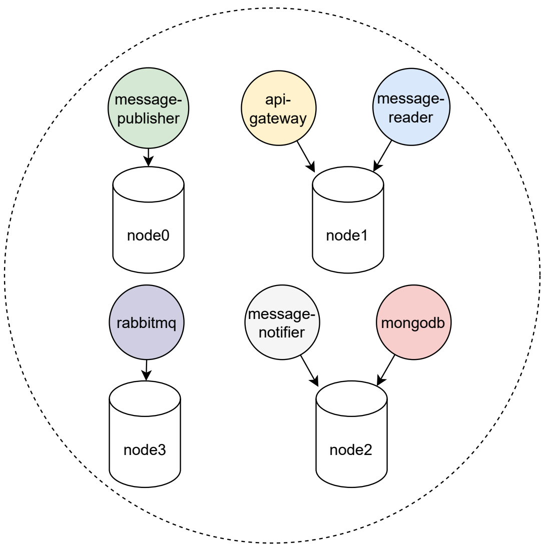

Figure 5: Sample application microservices placement.

At run-time, traffic amount exchanged between microservices is collected by the metrics server, in such a way that the custom plugin can use this information for future node scoring phases. Assuming that the *message-publisher* service receives a higher HTTP request load than that of *message-reader* service, the communication affinity between the *api-gateway* service and the *message-publisher* service becomes higher than the affinity between the *api-gateway* service and the *message-reader* service. If a new Pod for the *api-gateway* service is created (because for example of a failure of the actual Pod, a rolling update or an horizontal scaling), the custom plugin assigns a higher score to *node0* during the scoring phase of its scheduling cycle. This way, if the *api-gateway* Pod is placed on *node0*, mean response time experimented by external users when publishing messages is reduced.

6 CONCLUSIONS

In this work we proposed an extension of the default Kubernetes scheduler realized in the form of a custom scoring plugin. The proposed plugin takes into account application communication requirements, traffic history and network latency metrics to assign node scores when scheduling each application Pod. Application communication requirements are specified through a TOSCA service template that represents application components as node templates and microservice-to-microservice communication channels as relationship templates. Application traffic history and current node-to-node latency measures are instead queried from a Prometheus server. As a future work we plan to add the possibility to model more quantitative communication requirements, like deadlines on communication channels, inside TOSCA service templates and to design custom Kubernetes controllers that monitor application components and redeploy them if those requirements are not met.

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*.

Aral, A., Brandic, I., Uriarte, R. B., De Nicola, R., and Scoca, V. (2019). Addressing application latency requirements through edge scheduling. *Journal of Grid Computing*, 17(4):677–698.

Bulej, L., Bureš, T., Filandr, A., Hnětynka, P., Hnětynková, I., Pacovský, J., Sandor, G., and Gerostathopoulos, I. (2021). Managing latency in edge–cloud environment. *Journal of Systems and Software*, 172:110872.

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., and Wilkes, J. (2016). Borg, omega, and kubernetes. *ACM Queue*, 14:70–93.

Calcaterra, D., Di Modica, G., Mazzaglia, P., and Tomarchio, O. (2021). TORCH: a TOSCA-Based Orchestrator of Multi-Cloud Containerised Applications. *Journal of Grid Computing*, 19(5).

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).

Gedeon, J., Brandherm, F., Egert, R., Grube, T., and Mühlhäuser, M. (2019). What the fog? edge computing revisited: Promises, applications and future challenges. *IEEE Access*, 7:152847–152878.

Haghi Kashani, M., Rahmani, A. M., and Jafari Navimipour, N. (2020). Quality of service-aware approaches in fog computing. *International Journal of Communication Systems*, 33(8).

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.

Nastic, S., Pusztai, T., Morichetta, A., Pujol, V. C., Dustdar, S., Vii, D., and Xiong, Y. (2021). Polaris scheduler: Edge sensitive and slo aware workload scheduling in cloud-edge-iot clusters. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*, pages 206–216.

OASIS (2020). Topology and Orchestration Specification for Cloud Applications Version 2.0. http://docs.oasis-open.org/tosca/TOSCA/v2.0/TOSCA-v2.0.html.

Oleghe, O. (2021). Container placement and migration in edge computing: Concept and scheduling models. *IEEE Access*, 9:68028–68043.

Rodriguez, M. A. and Buyya, R. (2019). Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 49(5):698–719.

Rossi, F., Cardellini, V., Lo Presti, F., and Nardelli, M. (2020). Geo-distributed efficient deployment of containers with kubernetes. *Computer Communications*, 159:161–174.

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.

Salaht, F. A., Desprez, F., and Lebre, A. (2020). An overview of service placement problem in fog and edge computing. *ACM Comput. Surv.*, 53(3).

Tamburri, D. A., Van den Heuvel, W.-J., Lauwers, C., Lipton, P., Palma, D., and Rutkowski, M. (2019). Tosca-based intent modelling: goal-modelling for infrastructure-as-code. *SICS Software-Intensive Cyber-Physical Systems*, 34(2):163–172.

Toka, L. (2021). Ultra-reliable and low-latency computing in the edge with kubernetes. *Journal of Grid Computing*, 19(3):31.

Varshney, P. and Simmhan, Y. (2019). Characterizing application scheduling on edge, fog and cloud computing resources. *CoRR*, abs/1904.10125.

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.