Decentralized Node Distribution within a Federation of Kubernetes Clusters in a geo-distributed Fog Network

Victor Bieszka
Technical University Berlin
Berlin, Germany
v.bieszka@tu-berlin.de

Berit Frech
Technical University Berlin
Berlin, Germany
berit.frech@campus.tu-berlin.de

Christopher Woggon
Technical University Berlin
Berlin, Germany
christopher.woggon@tu-berlin.de

***Abstract*—Modern IoT application benefit from real-time data processing at the edge as it enables low latency while decreasing internet bandwidth usage. However, for smart city applications, edge devices are not only restricted and unreliable, but also geo-distributed. To cope with the vast management effort of a large and distributed network, it may be necessary to organise the network in sub-groups and handle resource allocation decentralised within the groups. We designed and implemented a solution that combines multiple Kubernetes clusters to a Federation. The within-cluster orchestration is handled by the inherent Kubernetes scheduler and it is extended by a decentralized node distribution among the clusters. The distribution of nodes among the clusters is running decentralized based on latency and current load and ensures that each cluster has enough resources to prevent long pending or failing requests. With this approach we were able to decrease overall roundtrip times while improving the success rate of requests.**

***Index Terms*—Serverless, Kubernetes Offloading, Multi-cluster, Fog Computing, Grouping**

I. Introduction

Serverless computing is an event-driven model, where applications use Function-as-a-Service (FaaS) platforms to run single functions without needing to take care of deployment, scaling mechanisms and maintenance [1]. Thus application logic is decoupled from the underlying infrastructure and code is executed on request rather than running permanently. This makes it easier to move parts of the applications and to allocate resources efficiently.

The serverless programming paradigm was enabled by the development from monolithic applications to microservices and event-driven programming on the developers' side and containerization and the pay-per-use model on the infrastructure providers' side [2].

By decoupling application logic from the underlying infrastructure, serverless computing enables developers to focus on high-level abstractions of their business logic (in the form of functions and events) that is mapped to the concrete execution environment by cloud providers who also handle orchestration, i.e. containerization, deployment and provisioning [3]. Consequently developers will benefit from this programming paradigm if they decompose their application logic into small units (or microservices) that can easily be containerized, deployed and provisioned on demand.

Serverless platforms like Apache OpenWhisk[1], OpenFaas[2] or AWS Lambda[3] were originally built for the Cloud, but properties like isolated execution, applications split into single functions, the event-driven execution, fine-grained scalability and the pay-per-use model are also making them a good fit for the edge [2] [4]. But edge devices usually have very limited resources. Microservices or functions that are provisioned on demand rather than long-running containers enable a higher and more efficient utilization of these scarce resources.

Additionally, handling computation at the Edge enables real-time data processing close to the end user and decreases internet bandwidth usage since less data needs to be transported towards the cloud [5]. This is beneficial for many IoT use cases, like smart homes, smart cities or autonomous driving, that rely on low latency.

In these use cases, the amount of transmitted data is increasing exponentially. The rise in city population and the increasing number of everyday life objects, that are turned into smart devices, are fostering this growth. Besides, mobility of end devices and unreliable network connectivity represent obstacles to computing solely at the Edge. Thus, to meet the demand of modern IoT applications, cloud and edge nodes should be combined to create a large fog[4] network that combines low latency computation at the edge with scalable, infinite resources in the cloud [6].

However, as the number of connected devices increases, the complexity of the system and the geo-distribution increases, too. This may result in an infeasible management effort in case of configuration changes or updates. Further, network latencies, failures or message loss becomes more frequent in large geo-distributed environments, making it harder to

[1] https://openwhisk.apache.org/
[2] https://www.openfaas.com/
[3] https://aws.amazon.com/de/lambda/
[4] Here, the definition of fog computing as the combination of cloud, edge and any intermediary node is used, while edge computing exclusively refers to computation at the edge of the network as in [6].

maintain the quality of services [6] and to obtain efficient energy-consumption and bandwidth-usage.

Thus, to make the fog network manageable and scalable, it may be necessary to organize it in autonomous groups. Within groups, orchestration and management of QoS becomes feasible again. However, to further benefit from all the available resources of the fog network, these groups should not be static, but dynamically adapt to current network conditions.

In the following we suggest to organize a fog network into multiple autonomous Kubernetes clusters that communicate with each other to distribute nodes according to current load and latency. We will evaluate this approach for a serverless use case.

In Section II we introduce the undelying technologies: Kubernetes, Kubernetes Cluster Federations and OpenFaas. The Problem Statement and Project Goal is specified in Section III. After an overview of related work in Section IV, we will describe our Concept and Design in Section V and the respective implementation in Section VI. Our Implementation is evaluated in Section VII and in Section VIII we will discuss our results and give some perspective on future work.

II. Background

A. Kubernetes

Kubernetes [7] is the most widely used open-source orchestration platform. It simplifies the orchestration of containerized application by handling deployment, resource scheduling, load balancing and server distribution in a large-scale infrastructure like cluster. It provides an abstraction from small application units, like functions or microservices, to the underlying infrastructure.

A Kubernetes Cluster contains a master nodes and any number of worker nodes. The worker nodes provide the environment for client applications while the master node provides the environment for the control plane, which is responsible for state management of the cluster. The control plane further provides API endpoints to communicate with the cluster, scheduler, and a data storage. Workloads in Kubernetes are run by placing containers into Pods that are run on nodes. Pods are a set of related containers and data volumes. A Node can be either a virtual machine or a physical device, depending on the use case. The assignment from workloads (Pods) to nodes is handled by the scheduler. Together, the Kubernetes components provide a highly available mechanism for failure recovery and an efficient load distribution within a single cluster.

However, the inherent Kubernetes scheduler only considers CPU and RAM and not latency, bandwidth usage [8] or locality [9]. In latency-sensitive IoT applications like smart driving, where fast reactions are crucial, this becomes impractical. Furthermore, Kubernetes was designed for local clusters in the cloud and hence relies on network connectivity and low latency between the nodes [10], which is not always given in a fog environment.

B. Cluster Federations

Multiple single cluster can be combined to form a cluster federation. In an IoT/Edge scenario a federation of autonomous clusters allows to meet the local requirements of geo-distributed applications and improves scalability. Due to the presence of one control plane per cluster, the reliability in case of network faults is improved in a cluster federation compared to a single cluster [11]. Reliability is further enhanced, if the cluster federation is spanned across multiple regions (geo-distributed) because network failures in one region can be absorbed by another region [12].

Additionally, maximum number of connected nodes for one Kubernetes cluster is 5000[5]. For a large smart city application, it is hence inevitable to increase the amount of connected nodes by forming a cluster federation.

Having a federation of clusters also opens possibilities to easily migrate workflows or applications from one cluster to another in case of disruptions or to scale and share resources among the clusters in case of imbalanced load.

C. OpenFaas

OpenFaas has been an emerging Serverless framework allowing developers to deploy functions and microservices with ease to Kubernetes. At it's core is the `OpenFaaS Gateway` which exposes all functionality - such as deploying and invoking functions and accessing metrics - via a REST API, CLI and UI. The second most important part of the architecture is the so called `faas-provider` interface that provides a CRUD API to actually invoke and scale up and down functions. The `OpenFaaS Gateway` directly communicates with the `faas-provider` through it's API. The two most commonly used `faas-providers` - and also utilized in this work - are `faas-netes` for Kubernetes and `faasd` for single host, no cluster OpenFaaS. Additionally, OpenFaaS runs Prometheus for monitoring and an AlertManager that tells the `OpenFaaS Gateway` to scale functions up if the load increases or to scale down if functions are running idle. OpenFaaS' popularity stems from it's low overhead when developing, running and maintaining functions. It takes care of exposing functions to a REST API, scales functions automatically according to the load and easily integrates with underlying container orchestration tools such as Kubernetes or K3s.

III. Problem Statement

Ideally all of the available resources of a fog network should be utilized efficiently. Particularly with a pay-per-use model, where resource utilization efficiency implies cost efficiency. This involves a mechanism to cope with fluidity of connected devices within the network.The network needs to adapt seamlessly, especially in a smart city environment, where mobile phones, cars or other moving smart objects are connected. Furthermore, applications that rely on low latency, like a traffic control system, need quick processing of the data at the edge. This is also required for sensitive data, that

[5]https://kubernetes.io/docs/setup/best-practices/cluster-large/

should not be transferred farther away from the data source than necessary. As many smart city application span across a large area, locality of nodes should also be incorporated in scheduling and placement decisions. Finally, for services like traffic control, availability is crucial. Thus, availability must always be given, even in the event of network failures.

A centralised management of the fog network would yield the most efficient resource allocation, as scheduling mechanisms can optimize with universal knowledge about the network. But this central component represents a bottleneck if all traffic needs to be directed towards the central component. Also, if services depend on a central component, availability may be reduced if network failures become more dominant, which is the case for large environments. Decentralized components even run when disconnected. Thus they reduce network bandwidth usage, latency at the edge and availability. Especially in geo-distributed networks, a decentralized architecture allows to incorporate locality of nodes to scheduling decisions. However, relying on a static, decentralized management of the network does not take imbalanced load patterns and network changes into account and therefore will not achieve overall efficiency. For example, a traffic control system may encounter bulk requests from a street with lots of traffic while somewhere else, little requests arrive. In such a scenario, a Kubernetes scheduler can have too many pending pods and thus response time will increase.

To achieve both, low latency at the edge and high availability, we use a hybrid solution with groups composed of Kubernetes clusters that communicate to exchange nodes. Kubernetes has already proven to reach efficient resource allocation and fault tolerance within a cluster. To mitigate the problem of pending pods, and thus increase availability, the clusters can exchange nodes to gain more resources. New nodes will be chosen decentralized among the clusters based on latency and the current load.

IV. Related Work

Splitting a fog network into groups based on latency was already proven a solution for reduced communication latency at the edge [13]. Within groups, broadcasting of messages yields little response time while selective message dissemination between groups reduces the message overhead, which is beneficial for latency-sensitive applications. In the following, we will apply the idea of latency-based grouping to node distribution rather than message dissemination.

Further work has been done on extending the default Kubernetes scheduler with network-aware properties, like latency and bandwidth consumption, to enable more informed resource allocation decisions [8]. This makes Kubernetes applicable to IoT applications in a fog environment. However, it only improves resource utilization within a single cluster and not among multiple clusters of a Federation.

Kubernetes Cluster Federation (KubeFed) [14] authorizes the coordination of multiple, geo-distributed Kubernetes clusters from a single control plane by extending the Kubernetes API. KubeFed was already tested in a decentralized fog environment and yields higher reliability and robustness compared to a single cluster setup [11]. Nevertheless, KubeFed is lacking support for dynamic cluster configuration changes and autoscaling of resources, as pod placement is static. Therefore KubeFed is lacking automation, making it difficult to apply for a dynamic fog environment with mobile edge devices and frequent node failures.

Liqo [15] is an open source tool that provides a multi-cluster control plane and enables resource sharing among different clusters. It enhances the properties of KubeFed, that only provides a control plane for synchronisation among clusters, with an interconnection of clusters. That connection is used to make services accessible by the other clusters of the federation. Resource sharing among clusters is handled via remote clusters that can be accessed and used by any cluster of the federation [16]. Yet, the remote resources are only 'rented' for computation of the offloaded workload and not completely assigned to a cluster, thus only handling short-term shortages.

To address KubeFed's lack of automation and variable scheduling settings, mck8s [17] was introduced. Mck8s is an extension on Kubernetes and KubeFed that provides a multi-cluster orchestration platform with the aim to maximize resource-utilization [18]. It offers a scheduler with several placement policies and horizontal pod-scaling over all clusters of the federation. Availability is enhanced with dynamic cluster provisioning capabilities and by adjusting the replica count according to current load.

In our approach we designed and implemented a dynamic cluster federation where nodes are exchanged on demand, rather than provisioning of new resources. Thus we distribute the existing nodes of a network depending on the current load while keeping the overall amount of resources stable. Our focus is on a decentralized node distribution among the federation, rather than sharing of resources and pod-scaling.

V. Concept and Design

A. Discourse: Initial Approach without Kubernetes

We will briefly describe our first approach to motivate the usage of Kubernetes for within-cluster orchestration.

With scalability in mind, we first implemented a solution with minimal memory footprint to reduce costs of the network by using 1GB virtual machines. In contrast, Kubernetes master nodes need at least 2GB of memory and worker nodes need at least 700MB to run without faults [19].

In order to reduce the memory footprint, our initial solution was built with faasd[6], a lightweight solution of OpenFaas for restricted edge devices that does not require a cluster. To make resource utilization efficient, we implemented scheduling, function placement and failure handling with faasd-functions that are executed on demand. For storage we used a SQLite[7] instance on each node (i.e. virtual machine).

[6]https://github.com/openfaas/faasd
[7]https://www.sqlite.org/index.html

To enable function chaining or function workflows, and to add a recovery mechanism in case of node failures, a utility module was added. This module enables functions to retrieve data from the SQLite instance and to extract their public IP-address. The IP is needed to call other faasd-functions from within a function, as they are running in containers and thus cannot call `localhost` to reach the `faasd-gateway`. To retrieve data from the SQLite instance, we further needed to implement a wrapper that enables access.

During the evaluation of this approach, the 1GB nodes were running out of memory, resulting in unpredictable behavior. Thus we had to increase memory. The minimum node on Google Cloud, where the project is running without errors, has 1 virtual CPU with 3.75 GB, enough memory to run Kubernetes.

Hence, we decided that it is preferable to use Kubernetes for within-cluster orchestration, as it is a widely adopted solution with a CPU- and RAM-usage optimizing scheduler and a reliable failure recovery mechanism. The updated approach with groups composed of Kubernetes clusters will be described in the following.

B. Kubernetes Approach

The initial setup of the Kubernetes-based grouping is composed by one cloud instance and multiple Kubernetes clusters that are geo-distributed and form a federation. With this approach, the within-cluster orchestration is completely left to the Kubernetes scheduler whereas the between-cluster organisation is handled decentralized among the clusters based on a cluster list that can be retrieved from the cloud instance.

Between-cluster organisation includes a load analysis within the cluster and a node exchange between clusters that is triggered based on a CPU-usage threshold. During load analysis, the current CPU of the nodes within a cluster is monitored. As soon as a certain threshold is exceeded, the cluster will nudge a node exchange process that is requesting more compute power (nodes) from other clusters of the federation to impede pending pods.

To initiate the node exchange, the requesting cluster will fetch a list with other clusters of the federation from the cloud instance and then compute latency to each cluster. To minimize latency, the requesting cluster will demand new compute power from the cluster with lowest latency values. A cluster that is receiving a node exchange request, will either suggest a node, if it has enough capacity, or will cancel the node exchange. As soon as a node exchange request succeeded, the node will be signed out of the initial cluster and registered at the requesting cluster.

To reduce the communication overhead during the second step of the node exchange, the list, that is retrieved from the cloud, does not include all members of the federation. The cloud instance pre-selects members based on the geo-location, as this is a first indicator of latency. Thus, in the second step, only latency to a limited number of clusters must be evaluated which reduces the bandwidth usage.

This method uses a central component (the cloud) which can represent a single point of failure for a network. However, the central instance is only used for the member-list retrieval and for remote function deployment, in case new functions need to be deployed to the network (that means, only functions that haven't been deployed before, in case of node failures or node exchange, the functions will be redeployed by the Kubernetes scheduler). Consequently, the functionality of the deployed services within the network do not rely on the central as the Kubernetes clusters will run independently, even when disconnected. Additionally, the central instance is running in the cloud, that means it has seemingly infinite resources and thus a high reliability.

Altogether, this approach enables a decentralized node distribution among the groups of the network. As the location and load of requests changes, the node distribution will adapt, too, thus preventing pending pods in the Kubernetes Cluster. Using Kubernetes for within-cluster orchestration enhances availability and latency at the edge, while making sure that the latency within the cluster is small, helps mitigating the lack of network-awareness of the inherent Kubernetes scheduler (see Section II).

VI. Implementation

A. Setup

To facilitate the deployment of the project, various Ansible playbooks have been created. These build and configure the necessary infrastructure in the Google Cloud, such as a compute instance for the cloud script and multiple Kubernetes clusters. Authentication is solved through the use of a service account.

As for the Ansible cloud playbook, the compute instance is updated, Python and all the utilized modules are installed, and the cloud Flask script is executed. The Flask script is providing the REST API. The cluster playbook simply sets up multiple clusters, each with a single node-pool consisting of three homogeneous nodes. These nodes are comprised of Google Cloud's `n1-standard-1` machines with 1 virtual CPU and 3.75GB of memory. Further cluster-specific configuration, such as configuring the firewall and the installation of OpenFaaS, is done by manually using `kubectl`, `gcloud` and `helm`. Afterwards, the docker containers, that are facilitating the cluster surveillance and node exchange, can be deployed from our Docker registry using Kubernetes deployments, which ensure that containers are always running and restarted in case of node failure, node shutdowns and pod eviction.

B. Cloud Instance

The primary purpose of the cloud instance is providing clusters with information regarding other registered clusters as well as remotely deploying OpenFaaS function workflows with a single request.

Clusters can register at the cloud instance, providing their IP and port, the respective OpenFaaS authentication data for remote deployment and their location, depicted by longitude

and latitude. Once registered, they can request a list of other registered clusters which are geographically close to them. The list of clusters contains their addresses and is returned sorted by distance. The number of returned clusters can be limited by a parameter.

When remotely deploying a function workflow through the cloud, the target cluster IP as well as a list of function names and associated registry URLs are passed to the `deploy-functions` cloud endpoint. Deployment will be handled entirely independent in the targeted cluster.

C. Cluster

The cluster code has been separated into two docker containers – one called `kubectl-gcloud` which handles all interactions with the Kubernetes cluster and the Google Cloud via `kubectl` and `gcloud`. The second container, called `cluster-app`, utilizes endpoints exposed by the first container in order to survey the cluster and exchange nodes with other clusters if necessary.

Endpoints provided by the first container include `/get-node-info`, which provides the current load of all nodes in the cluster and `/add-node`, which adds a node to the cluster's node-pool, thus increasing it's size by one. The last endpoint, `/delete-node`, first cordons a node. This ensures that no new pods can be scheduled on this node by Kubernetes, thus avoiding inefficient pod creation. Afterwards, the node is drained. During drainage, all the running pods are evicted and rescheduled on the remaining nodes in the cluster. As soon as the eviction process is finished successfully, the node is removed from the cluster and the instance-group it belonged to. Leaving the node in its former instance-group would lead to the Google Cloud automatically rescheduling the node to its initial state and thus leaving it in the cluster that it was meant to be removed from. To finalize the node exchange process, the node is removed from the Kubernetes cluster. Afterwards, requests are again authorized with the use of a Google cloud service account. Additionally, a Kubernetes service of type `ClusterIP` allows other pods to communicate with this pod internally.

The second container, after it was initially registering at the cloud, regularly checks the current cluster state. As soon as the current load exceeds a configurable threshold, it will request other clusters' information from the cloud. By pinging other clusters, it re-sorts the clusters by actual latency before requesting a node exchange. The node-exchange requests are sent starting from the cluster with the lowest latency. The sorted list is traversed until until one cluster is found that can continue operation with one node less. While the responding cluster agrees to shutdown a node, the requesting cluster will add a new node to the cluster. During this process it is not the exact same node that gets passed from one cluster the other. However, this is the closest approximation to a node exchange that one can build while utilizing the google cloud tools which are available. Once the start-up of the node has been initialized, further addition or removal of nodes is disabled until the new node is fully set up and added to the cluster.

Alternatively, a node can be requested from the cluster through the `request-node-exchange` endpoint. If the node exchange is not currently locked, the average load of the cluster will be determined. If removing a node will not push the load over a configurable threshold, the currently least utilized node will be shutdown while a new node will be added to the distant cluster. For Kubernetes to continue running, the minimum cluster size is one node. Hence, at this point, further adding or removing of nodes is disabled for three minutes.

Generally, important steps in the re-balancing process will be logged and can be accessed through the `get-log` endpoint.

VII. Evaluation

In this section we describe the results of the evaluation in order to outline the added value of our implementation.

Figure 1 displays different properties of two Kubernetes clusters plotted over time. Specifically, the average CPU load and number of nodes for both clusters. In this scenario, the first cluster was consistently challenged by thousands of requests to an OpenFaaS function using the `hey`[8] commandline tool. During the experiment, roughly 17000 requests were sent. The Python OpenFaaS function *ping-pong* returns the word *pong* upon invocation.

As is evident from the graph that the initial roundtrip time averaged over 4s. This is likely due to the fact that initially there was only one replica and OpenFaaS took some time to increase the number of replicas to utilize all three cluster nodes. In this graph, bundles of 100 request roundtrip times are averaged, thus individual roundtrip times might be significantly higher. The average CPU load of the cluster also increased to above 40% at this time. This likely happened due to one node being fully loaded. The delay, that is evident from the graph, can be attributed to the fact that the node CPU load is updated quite infrequently. Thus, the first node probably was under full load from the get go.

The CPU load further increases up to almost 100%. At this time, the node exchange, which was initiated once the average CPU load exceeded 50%, has been finalized. The sudden drop to roughly 60% CPU utilization can partly be explained by the new, not yet utilized, node being included in the CPU load calculation. If all other 3 nodes were at full load, we would reach an average load of a little more than 75%, assuming that the new node is at 0% CPU-usage.

At this point, the CPU load continues to increase again, almost reaching 100% load again. Since the sudden drop to 60%, OpenFaaS function replicas have been deployed on the fourth node.

Even a little earlier, about 250s into the experiment, the second cluster has finished shutting down another node, after previously agreeing on moving another node. At roughly 325s into the experiment, adding the node to the first cluster has been finalized. It resulted in a slow decline in average CPU utilization of the first cluster, back to under 60%. At this point,

[8] github.com/rakyll/hey

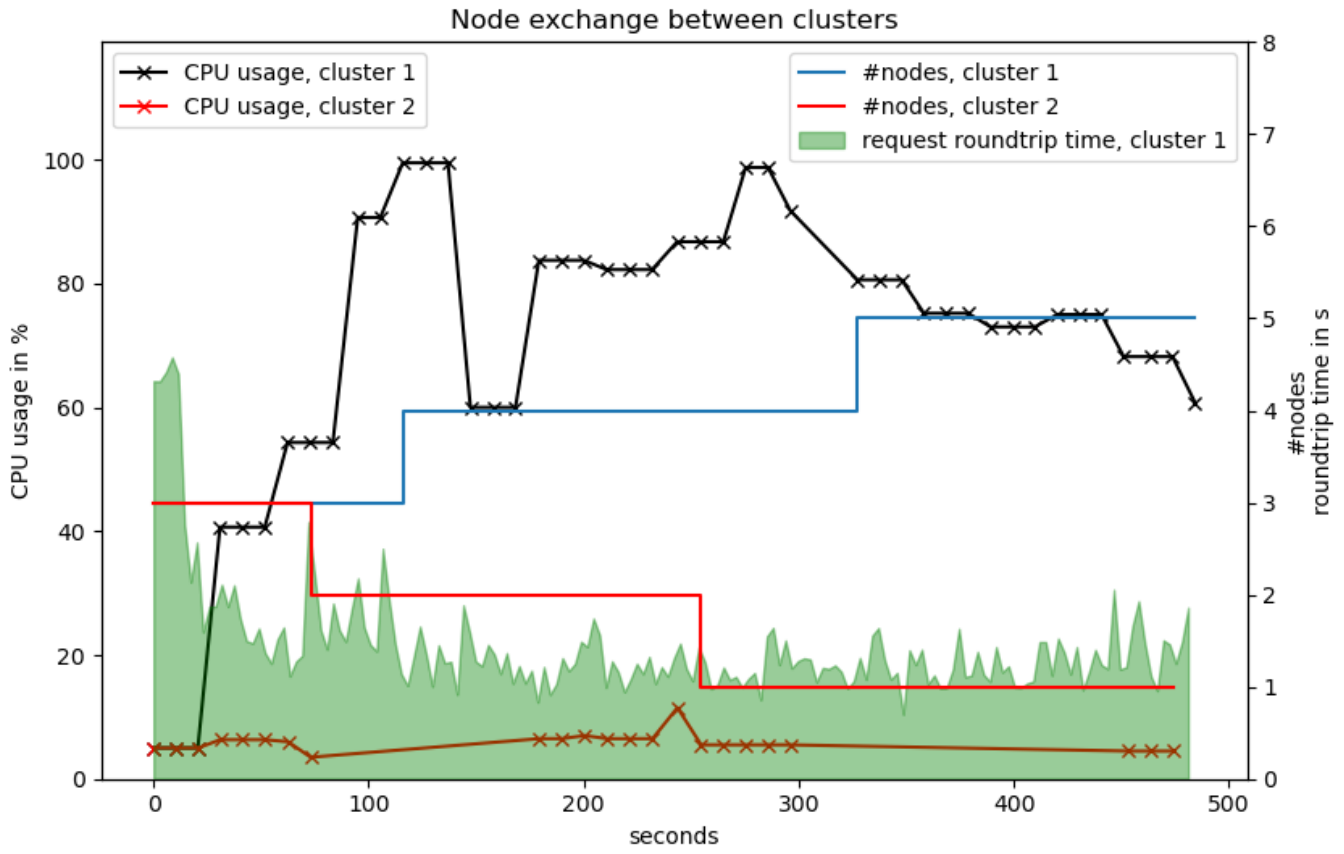

Fig. 1. Node Distribution among two Clusters with bulk request to Cluster 1

as the second cluster now consists of only one node, further moving of nodes to the first cluster is no longer possible.

Roundtrip times had almost exclusively been under 2s since the fourth node was added to the first cluster. From that point on, there weren't any notable changes, even after adding a fifth node. This stabilization is to be somewhat expected with CPU load not reaching 100%.

The CPU load of the second cluster is consistently below 15%. Overall, about 5% of requests were answered with non 200 status codes.

Figure 2 displays an experiment with the same cluster composed of three nodes, but with the node exchange functionality disabled. It is apparent that roundtrip times are much greater in this scenario. In the graph, the vertical green lines represent requests that had been answered with a 200 status code. In total, over 96% of response status codes were either `500 Internal Server Error` or `503 Service Unavailable`. In total, only 1227 requests have been sent, a much smaller number than in the first experiment. The reason for this is the significantly longer roundtrip time while having the same experiment runtime. To create an informative plot with much fewer datapoints, bundles of 10 request roundtrip times have been averaged, as opposed to 100 request roundtrip times in Figure 1.

The evaluation clearly shows that our approach managed to lower roundtrip times and increase request success rates significantly.

VIII. Conclusion and Future Work

In this work, we designed and implemented a grouping of a network into multiple Kubernetes clusters that reduces roundtrip times and increases availability by exchanging nodes. Our approach ensures that enough resources are available in each single cluster at all times by defining a CPU-load threshold that triggers a node-exchange process. During this process, a cluster will request new compute power from other clusters of the federation based on latency. Once a request is accepted, the node will be moved from one cluster to the other. Hence, cluster sizes adapt dynamically to cope with the irregular and imbalanced load distribution among the network. Geo-distributed applications can benefit from this approach as it incorporates locality of nodes, reduces response time at the edge and allows for a better resource distribution.

We evaluated our approach in a simplified setting with two Kubernetes Clusters running on google cloud. By sending bulk requests to one of the clusters, the experiments showed that our mechanism is working. The results confirmed that the dynamic exchange of nodes between the clusters indeed reduces latency and increases the success rate of requests.

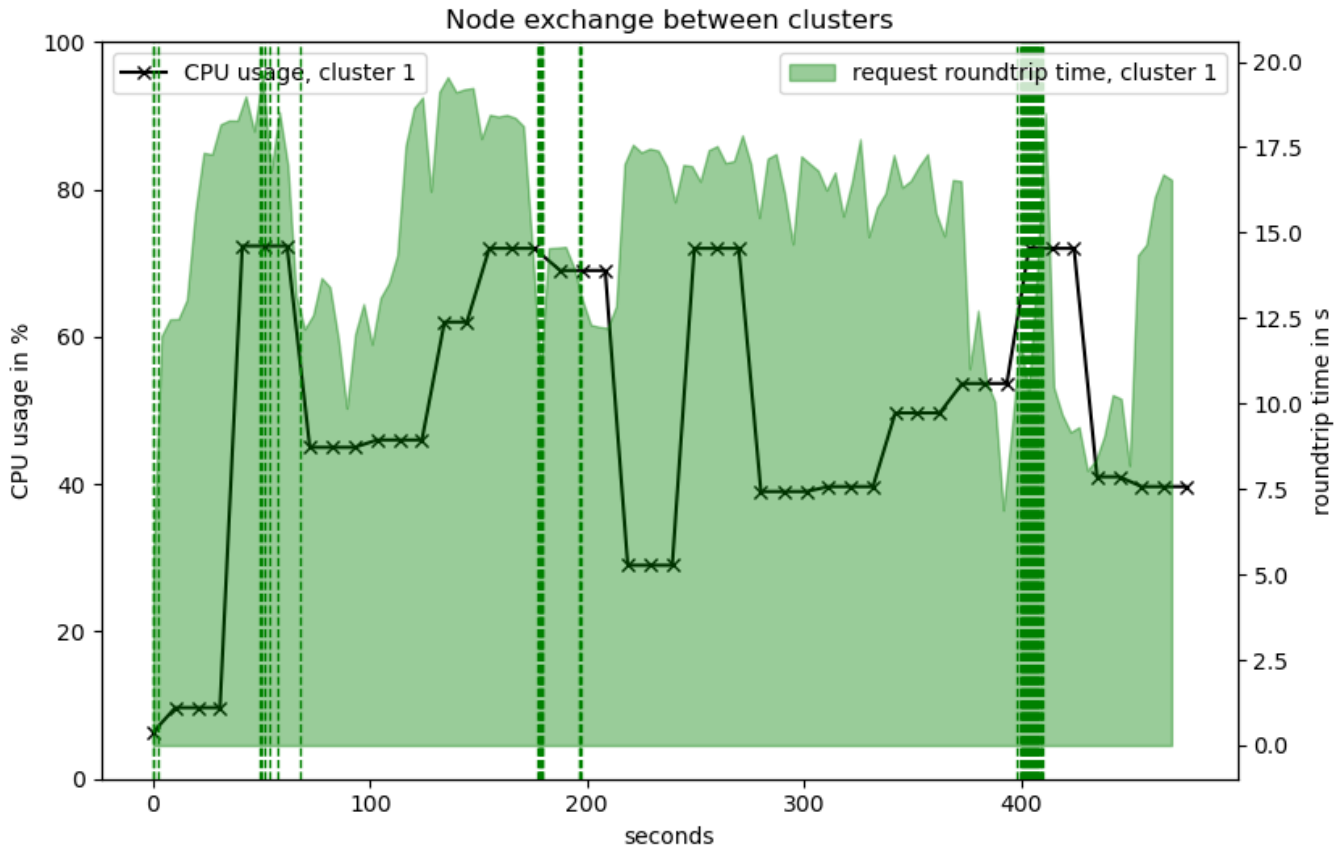

Fig. 2. Bulk requests to single cluster

To further decrease latency of requests, the Kubernetes scheduler can be extended with network-aware properties for within the cluster and not only consider CPU and RAM usage [8]. Additional improvement can be achieved with a decentralized control plane [20]. Thus, substituting the cloud instance with a cluster enhances availability and scalability of the approach. Finally, we evaluated our approach with a small cluster setting and only virtual resources. Therefore the approach should be tested in a large and geo-distributed network with virtual and physical devices, like raspberry pies, to have a more realistic IoT setting where physical devices could be installed on traffic lights, cars or sensors. Another step would be going a middle way between the initial approach - using `faasd` in combination with no clustering - and the final approach with a fully fledged OpenFaaS and Kubernetes implementation. One such tool could be K3s[9], a lightweight Kubernetes distribution built to be run on IoT and Edge devices. This could further reduce CPU and memory consumption bringing it closer to the initial `faasd` idea while also providing all the benefits that come from using clusters.

[9]https://k3s.io/

REFERENCES

[1] G. McGrath and P. R. Brenner, "Serverless computing: Design, implementation, and performance," in *2017 IEEE 37th International Conference on Distributed Computing Systems Workshops (ICDCSW)*, 2017, pp. 405–410.

[2] M. S. Aslanpour, A. N. Toosi, C. Cicconetti, B. Javadi, P. Sbarski, D. Taibi, M. Assuncao, S. S. Gill, R. Gaire, and S. Dustdar, "Serverless edge computing: Vision and challenges," in *2021 Australasian Computer Science Week Multiconference*, ser. ACSW '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3437378.3444367

[3] E. van Eyk, L. Toader, S. Talluri, L. Versluis, A. Uță, and A. Iosup, "Serverless is more: From paas to present cloud computing," *IEEE Internet Computing*, vol. 22, no. 5, pp. 8–17, 2018.

[4] A. Glikson, S. Nastic, and S. Dustdar, "Deviceless edge computing: Extending serverless computing to the edge of the network," in *Proceedings of the 10th ACM International Systems and Storage Conference*, ser. SYSTOR '17. New York, NY, USA: Association for Computing Machinery, 2017. [Online]. Available: https://doi.org/10.1145/3078468.3078497

[5] L. U. Khan, I. Yaqoob, N. H. Tran, S. M. A. Kazmi, T. N. Dang, and C. S. Hong, "Edge-computing-enabled smart cities: A comprehensive survey," *IEEE Internet of Things Journal*, vol. 7, no. 10, pp. 10 200–10 232, 2020.

[6] D. Bermbach, F. Pallas, D. G. Pérez, P. Plebani, M. Anderson, R. Kat, and S. Tai, "A research perspective on fog computing," in *Service-Oriented Computing – ICSOC 2017 Workshops*, L. Braubach, J. M. Murillo, N. Kaviani, M. Lama, L. Burgueño, N. Moha, and M. Oriol, Eds. Cham: Springer International Publishing, 2018, pp. 198–210.

[7] Kubernetes. [Online]. Available: https://kubernetes.io/

[8] J. Santos, T. Wauters, B. Volckaert, and F. De Turck, "Towards network-aware resource provisioning in kubernetes for fog computing applica-

tions," in *2019 IEEE Conference on Network Softwarization (NetSoft)*, 2019, pp. 351–359.

[9] A. J. Fahs, G. Pierre, and E. Elmroth, "Voilà: Tail-latency-aware fog application replicas autoscaler," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020, pp. 1–8.

[10] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, "Instability in geo-distributed kubernetes federation: Causes and mitigation," in *2020 28th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2020, pp. 1–8.

[11] F. Faticanti, D. Santoro, S. Cretti, and D. Siracusa, "An application of kubernetes cluster federation in fog computing," in *2021 24th Conference on Innovation in Clouds, Internet and Networks and Workshops (ICIN)*, 2021, pp. 89–91.

[12] N. Grozev and R. Buyya, "Inter-cloud architectures and application brokering: taxonomy and survey," *Software: Practice and Experience*, vol. 44, no. 3, pp. 369–390, 2014. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2168

[13] J. Hasenburg, F. Stanek, F. Tschorsch, and D. Bermbach, "Managing latency and excess data dissemination in fog-based publish/subscribe systems," in *2020 IEEE International Conference on Fog Computing (ICFC)*, 2020, pp. 9–16.

[14] Kubernetes cluster federation. [Online]. Available: https://github.com/kubernetes-sigs/kubefed

[15] LiqoTech. Liqo. [Online]. Available: https://github.com/LiqoTech/liqo

[16] G. Arbezzano and A. Palesandro. Simplifying multi clusters in kubernetes. [Online]. Available: https://www.cncf.io/blog/2021/04/12/simplifying-multi-clusters-in-kubernetes/

[17] mkc8s. [Online]. Available: https://github.com/moule3053/mck8s

[18] M. A. Tamiru, G. Pierre, J. Tordsson, and E. Elmroth, "mck8s: An orchestration platform for geo-distributed multi-cluster environments," in *ICCCN 2021 - 30th International Conference on Computer Communications and Networks*, Athens, Greece, Jul. 2021, pp. 1–12. [Online]. Available: https://hal.inria.fr/hal-03205743

[19] Kubernetes cluster requirements. [Online]. Available: https://docs.kublr.com/installation/hardware-recommendation/

[20] L. Larsson, H. Gustafsson, C. Klein, and E. Elmroth, "Decentralized kubernetes federation control plane," in *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*, 2020, pp. 354–359.