

Kubernetes and the Edge?

Karim Manaouil, Adrien Lebre

▶ **To cite this version:**

Karim Manaouil, Adrien Lebre. Kubernetes and the Edge?. [Research Report] RR-9370, Inria Rennes - Bretagne Atlantique. 2020, pp.19. hal-02972686v2

HAL Id: hal-02972686
https://inria.hal.science/hal-02972686v2

Submitted on 22 Oct 2020

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

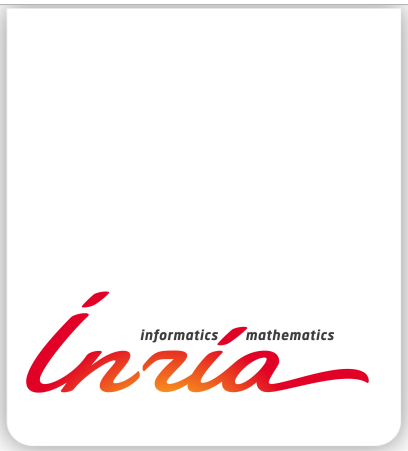


Kubernetes and the Edge?

Karim Manaouil , Adrien Lebre

RESEARCH REPORT
N° 9370
Octobre 2020
Project-Team Stack

ISSN 0249-6399 ISRN INRIA/RR--9370--FR+ENG



Kubernetes and the Edge?

Karim Manaouil *, Adrien Lebre *

Project-Team Stack

Research Report n° 9370 — Octobre 2020 — 19 pages

Abstract: Cloud Computing infrastructures have highlighted the importance of container orchestration software to manage the life cycle of distributed applications. With the advent of Edge Computing era, DevOps expect to find features that made the success of containerized applications in the cloud, also at the edge. However, orchestration systems have not been designed to deal with resources geo-distribution aspects such as latency, intermittent networks, locality-awareness, etc. In other words, it is unclear whether they could be directly used on top of such massively distributed infrastructures or whether they must be revised. In this paper, we provide reflections regarding Kubernetes, the well-known container orchestration platform. More precisely, we provide two contributions. First, we discuss results we obtained during an experimental campaign we made to analyze the impact of WAN links on the vanilla Kubernetes. Second, we analyze ongoing initiatives that propose to revise part of the Kubernetes design to better address geo-distribution aspects.

Key-words: Edge Computing, container orchestration, Kubernetes

* Inria, IMT Atlantique, LS2N

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Kubernetes et l'informatique de périphérie?

Résumé : Les infrastructures de Cloud Computing ont mis en evidence les technologies de type conteneur et notamment des logiciels d'orchestration qui facilitent grandement la gestion du cycle de vie des applications distribuees. Avec l'avenement de l'informatique en peripherie (fog/edge computing), il est fortement envisageable que les ingénieurs DevOps attendent de trouver ces memes fonctionnalites dans ces nouvelles infrastructures. Cependant, les systemes d'orchestration disponibles n'ont pas ete concus pour traiter les aspects de geo-distribution des ressources tels que la latence, les reseaux intermittents, la localite, etc. En d'autres termes, il n'est pas certain qu'ils puissent etre utilises directement sur des infrastructures massivement distribuees au travers plusieurs sites géo-graphiques. Dans cet article, nous presentons une étude préliminaire autour det Kubernetes, le standard open-source de-facto d'orchestration. Plus precisement, nous fournissons deux contributions. Tout d'abord, nous discutons plusieurs resultats obtenus lors d'une campagne experimentale que nous avons menee pour analyser l'impact des liens WAN sur le code de Kubernetes natif. Deuxiemement, nous analysons les initiatives en cours qui revisitent la solution vanilla afin de mieux aborder les aspects de geo-distribution. Si ces approches pourraient etre appropriees pour certains cas d'utilisation, elles sont malheureusement incompletes pour d'autres et il convient donc de proposer de nouvelles approches.

Mots-clés : informatique en périphérie , orchestration de conteneurs, Kubernetes

1 Introduction

Deploying computational and storage resources as close as possible to the end users has been debated during almost ten years due to economical reasons. With the advent of the Internet of Things based applications, the question, now, is no more whether they will be deployed but rather how operators and developers (DevOps) can administrate and respectively use them?

Two years ago, we underlined the importance of delivering a cloud-like resource management system for edge infrastructures [15]. Such a system should provide features that made current cloud computing solutions successful. Although significant progress have been achieved from the operators side, in particular to manage the life cycle of data-center infrastructures in a more automatized manner (Airship [10], Cluster API [11], etc.), only a few initiatives have investigated whether and how container orchestration systems should be revised in order to deal with geo-distribution aspects of Edge Computing platforms (location awareness, latency/throughput constraints, intermittent networks, etc.). This lack of studies comes as a surprise as orchestration systems are nowadays the recommended way to manage the full life cycle of distributed applications. The almost unanimous adoption of the Kubernetes framework is probably the best example. Since 2016, Kubernetes has been continuously proving its place in the Cloud market to become the defacto standard to deploy and manage distributed applications on top of major Cloud Computing platforms [21, 25].

Delivering a container orchestrator *à la* kubernetes is critical to favor the transition to the Edge Computing paradigm. However building a system from scratch looks rather impractical due to the complexity and associated overhead in terms of development. In this article, we propose to analyze the kubernetes ecosystem in order to highlight pros and cons of current solutions and identify challenges our community should address. To the best of our knowledge, this is the first study of Kubernetes under the geo-distributed perspective.

Since there are as many edge infrastructures as use-cases, we highlight that the infrastructure considered in this study is composed of several geo-distributed micro DCs composed of up to one hundred servers. To operate such WANWide infrastructures, two deployment scenarios are generally proposed: centralized or federated approaches [17]. The former lies on considering an edge infrastructure as a traditional single data center environment, the key difference being the wide-area network found between the service and worker nodes. The latter consists in deploying one instance per micro DCs and federating them through a brokering approach in order to give the illusion of a single global system. Unfortunately, Kubernetes has not been developed with one of the two scenarios in mind. At the opposite, it has been designed to hide the complexity of a single location cluster (LANWide) and in a pretty-standalone way (it cannot peer with other Kubernetes instances without additional pieces of software). To deal with such limitations, several initiatives are under development. However, they investigate different directions without really arguing their choices and without first evaluating the possible limitations of the vanilla Kubernetes . The goal of our study is to clarify these aspects. Concretely, we propose two contributions:

- First, we analyze how Kubernetes behaves WANWide by discussing in-vivo experiments we made on top of Grid'5000. Although its was not its initial target, the REST-based architecture of Kubernetes enables it to scale to multiple locations theoretically. With these experiments, our objective was to evaluate practically the impact of the latency on the control services.
- Second, we describe the main initiatives that propose to extend Kubernetes in order to deal with edge specifics, namely the KubeEdge [26], Kubefed [4], and Submariner [8] proposals.

Conducting such a study is important and, we believe, timely as several projects have chosen

to expose their edge infrastructures through Kubernetes (the EdgeNet project [14]) without identifying possible limitations a priori.

The remainder of the paper is organized as follows. Section 2 gives an overview of the Kubernetes concepts and its architecture. Section 3 studies how Kubernetes behaves WANWide, highlighting some limitations that we did not expect. Section 4 discusses open-source ongoing initiatives to deal with geo-distributed aspects. Finally, Section 5 concludes the paper and gives somes perspectives to this study.

2 Kubernetes

Kubernetes is a container orchestration platform that was initially developed by Google from lessons learned in a decade of running the Borg job manager [13]. It is primarily designed as a distributed and a scalable system to automate the life cycle of container-based applications on a cluster of machines. It provides a set of high-level abstractions to help DevOps easily deploy their applications, automatically scale in/out a particular service based on its load, perform rolling updates without inducing offline times, do DNS based service discovery, manage application configuration and perform a plethora of other services. Kubernetes is a growing ecosystem with rich functionality thanks to its design principles and its inherent extensibility.

We present in the following, the principles and concepts of Kubernetes (knowing them is important to understand the rest of the article). However, this part can be skipped for those who are familiar with the framework.

2.1 Kubernetes Design Principles

Kubernetes is built around a set of principles that leads to its design and architecture. At the heart are declarative configuration, immutable infrastructure and online self-healing. The central idea behind the declarative configuration is to let the system act on behalf of the user intent: The DevOps describes the desired state of the cluster and Kubernetes sits in an endless loop (reconciliation loop) trying to drive the system to the desired state continually. This involves fetching container images from central repositories, allocating and mounting storage spaces, configuring networks, configuring, starting, scaling and deleting containers, etc.

In Kubernetes, system changes are not performed incrementally. Although technically possible, it is considered as an anti-pattern to imperatively mutate the state of a container. Instead, container immutability is preferred by encouraging the operators to rebuild their images and restart their containers. In fact, this principle does not only apply to containers but all the types of Kubernetes objects. We present in the next paragraph, the most essential ones.

2.2 Kubernetes Objects

Kubernetes provides abstractions and high level concepts to manage deployed micro-services as generic as possible in order to perform various operations (discovery, scaling, load-balancing, rolling updates, etc.).

Pods a Pod is the basic element of operations in Kubernetes. Technically speaking, a Pod is a collection of containers that share the same process and network address spaces (virtual network interfaces, ports, IP addresses and shared memory). Processes inside a Pod can communicate with each other using sockets listening on localhost or with inter-process communication facilities such SysV shared-memory.

Labels and selectors: labels are key-value pairs used to keep track of objects inside a Kubernetes cluster. All objects in Kubernetes can be labeled. A DevOps can query to find all the

objects with a particular label or issue a command that will affect all of those objects in a single shot. Selectors are the complementary concept that select objects based on a set of given labels.

Services: a service provides load-balancing and DNS-based naming to applications running on the cluster. There are many types of services. In a *ClusterIP* service, a static virtual IP address is used as an `iptables` target. All requests to this target are forwarded to one of the corresponding Pods. The load-balancing is a feature of the netfilter subsystem of the Linux kernel. On the other hand, a *NodeIP* service uses the host IP instead. The service IP and port are provided as environment variables once the container is started and the service name is resolvable using the DNS server provided by Kubernetes (`kube-dns`).

ReplicaSets: a *ReplicaSet* is responsible of maintaining the desired number of replicas of a particular Pod. ReplicaSets are the best example of a reconciliation loop, they keep watching the cluster's state and schedule new Pods or remove existing ones in response to scalability demands or cluster failures.

ConfigMaps and Secrets: they are key-value objects stored in Kubernetes. *ConfigMaps* provide configuration information to the workloads. They are an essential part to make Pods reusable components. They can be mounted as a file (keys map to files and values map to files contents) or provided as environment variables. For example, an Apache or a Nginx Pod can be contextualized by providing the server's configuration as a *ConfigMap* mounted as a file. *Secrets* are a mean to provide sensitive information to the container like database credentials at container creation time instead of being stored on the container's file-system image.

Deployments: a Deployment is a higher level abstraction and it is the preferred API object to deploy a micro-service on Kubernetes. Besides managing a ReplicaSet, it also provides features to effectively perform application upgrades. More specifically RollingUpdates [6] that ensures the application Pods are smoothly replaced without inducing an offline time.

Namespaces: a Namespace is an abstraction that defines a logical scope for resources. They can be used to implement multi-tenancy and resource isolation between development teams or to create multiple virtual clusters on top of a single infrastructure.

2.3 Kubernetes Architecture

As mentioned, Kubernetes is a container as a service platform that is deployed on top of a cluster. The latter is a group of machines providing compute, storage and networking resources to deploy and run containerized applications. A Kubernetes cluster is comprised of a control plane and a set of worker nodes. The control plane is called the master and is responsible for managing the rest of the infrastructure. The workers are the nodes on which the objects are deployed (e.g Pods). Kubernetes is designed around a RESTful HTTP interface where the objects API and the different services exposed by each component are served using HTTP endpoints. Figure 1 depicts its architecture.

The master: It is responsible for scheduling Pods, managing replication, orchestrating rolling updates, hosting the objects database and maintaining the api-server which is the heart of the communication in Kubernetes. The master components can either be deployed on a single node or replicated and distributed across a set of nodes for high availability. It has the following components:

- **Etcd** is a highly-consistent, distributed key-value store that hosts the cluster's state and the all the API objects. In Kubernetes, objects stored in etcd are solely accessed through the api-server [3].
- **The Scheduler** is a plug-able component that is in charge of mapping Pods to nodes to run them. The default scheduler performs filtering through label selection then finds the best

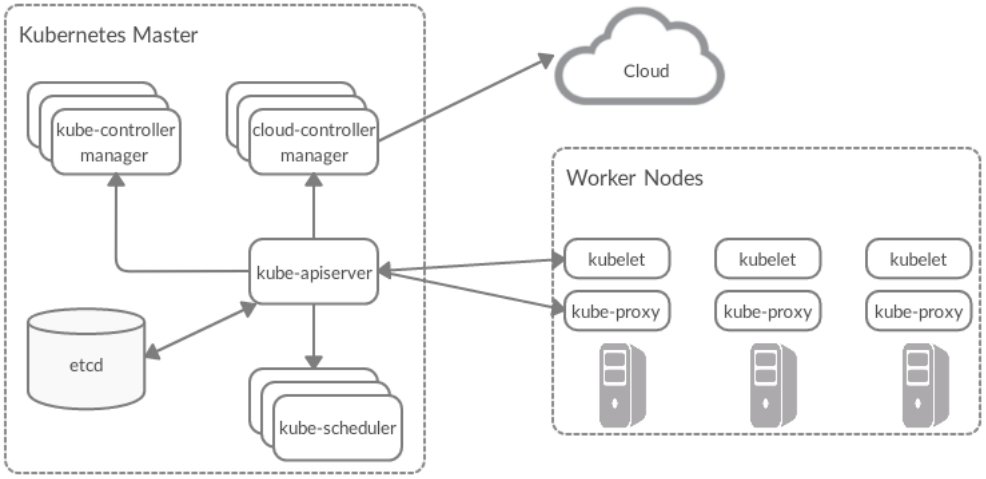


Figure 1: The architecture of a Kubernetes cluster

node that matches the Pod's specifications in terms of CPU, memory and disk capacity. The default scheduler runs in two phases, filtering and scoring. The filtering finds feasible nodes by applying filters such as the *PodFitsResources* and the *PodMatchNodeSelector*. The scoring step ranks the accepted nodes and chooses the highest ranked node to run the Pod such as the *BalancedResourceAllocation* strategy which only chooses the nodes with balanced resource usage [1].

- **The ControllerManager** is an aggregator that manages the different controllers on Kubernetes. The most fundamental concept of the controllers is the reconciliation loop in which they continually keep watching for object changes and then react to them in order to drive the cluster to the desired state. As an example, the replication controller is in charge of maintaining the number of replicas of a particular service as defined by the DevOps (notion of *ReplicaSet*).

- **The api-server** is the gateway and the glue that connects the distributed components together. As mentioned, it receives HTTP RESTful requests, validates the objects and pushes them to etcd. It also provides the front-end to the cluster's shared state through which all the other components interact. DevOps can interact with the API-server either by invoking directly the HTTP API or through a CLI wrapper, entitled `kubectl`.

The workers: they provide the resources to deploy and run the containerized workloads. There are two main components running on every worker node:

- **Kubelets** are the agent that represent the operational side of Kubernetes. they continuously watch for object changes in the cluster and react upon them. Their responsibilities include but not limited to: Pod creation and destruction, contextualizing Pod environments, mounting volumes in the containers file-systems, managing Pods networking and materializing secrets and ConfigMaps (e.g provide them as environment variables or mount them as file-system objects). They also updates and reports back Pods status to the api-server.

- **Kube-proxies** are responsible for managing service discovery on the nodes by creating forwarding rules with virtual IP targets that point to the Pods behind a service object (i.e

service endpoints). It also provides a way to do TCP and UDP streaming (e.g to implement port-forwarding and `kubectl`

3 WANWide Kubernetes

A simple way to expose multiple locations under the same Kubernetes cluster is to follow the centralized control plane pattern. In this approach, the Kubernetes master is deployed on one central location while the workers are deployed throughout the different edge sites. Figure 2 depicts such an architecture. Although the HTTP RESTful design of Kubernetes allows theoretically such a deployment scenario, there is no performance analysis of such a model. To fill this gap, we implemented a complete experimental artifact and conducted an experiment campaign on top of Grid'5000 [12]. We present in this section major results.

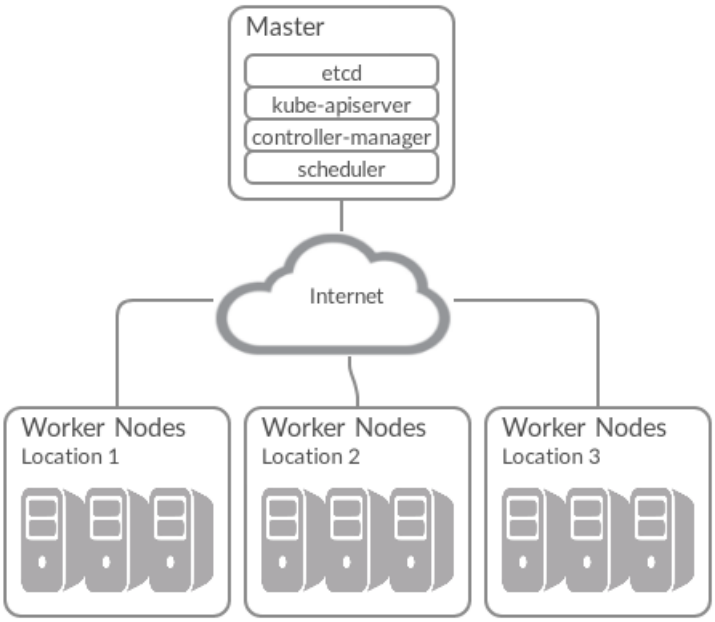


Figure 2: A WANWide Kubernetes cluster architecture

3.1 Experimental protocol

Evaluating a complex piece of software such as Kubernetes at WAN scale is an overwhelming task. To mitigate our efforts and allow other researchers to replicate and perform additional experiments, we developed a dedicated artifact similarly to what has been done for OpenStack [16].

Entitled `enos-kubernetes` [1], the artifact has been developed using `enoslib` [2], a library to build experimental frameworks on various testbeds. `enos-kubernetes` allows researchers to book resources on a particular testbed, auto-provision a Kubernetes cluster using Kubespray [5], perform a benchmark and collect mutiple performance metrics. Based on the `enoslib` deployment

[1] https://pypi.org/project/enos-kubernetes/
[2] https://discovery.gitlabpages.inria.fr/enoslib/

manifest, `enos-kubernetes` also contextualizes the infrastructure prior to performing the experiments. In our particular case, it applies packet delays and packet loss on the node's network interfaces through the Netem service [19] in order to emulate WANWide link.

Regarding the benchmark, we used ClusterLoader [2] that has been designed to perform a series of object operations (create a Pod, delete a service, scale a deployment, etc.). Those operations are described through a YAML-based descriptive manifest divided into multiple phases. Each object within a phase references a Kubernetes object template[3] as well as an optional Namespace. Operations described in a phase are performed concurrently. To enable the execution of multiple phases in an iterative manner, ClusterLoader allows the grouping of phases into steps. Steps are run sequentially (the next step won't start until all the phases of the previous step are finished). Phases can reference predefined *TuningSets*, which allow a fine grained control over the way the operations of a phase are performed across a time interval Another central concept of ClusterLoader is measurements. They allow to monitor and gather profiling data on a certain particular Kubernetes service level objectif (API responsiveness, Pod startup time, DNS latency, etc.). They are registered with a unique name with which they can be referenced from the benchmark manifest and they implement an interface of two methods named `Start()` and `Gather()`. The former is called to initialize and begin the monitoring. The latter is used to gather the data and finish the measurement (locate the Pod that contains the monitoring software, copy its database from the Pod to ClusterLoader's file-system then delete the Pod from the cluster). Other special-purpose measurements are provided to control concurrency and synchronize between events (wait until the number of replicas reaches a certain value before creating the next object).

The implementation of the measurements is let to DevOps. However, as Kubernetes provides a complete monitoring framework and abstraction layer that exports metrics in Promotheus format [7]. a straightforward way to implement measurements is to rely on them. In that sense, ClusterLoader deploys a Prometheus server that collects the different metrics exposed by Kubernetes (in particular the API requests latency).

Unfortunately, we had to perform two important changes. The first one is related to the API request latency reported by Kubernetes that only considers the time spent on the API server and not the time spent on the network also. To fix this issue, we implemented a new measurement directly on the GoLang HTTP client library against which all the components are compiled. This enables us to track the duration of a complete request (that is between the emission and reception of the HTTP request). The second change is related to the way the monitoring service arranges the requests. The original implementation groups the collected durations by HTTP *resource* and *verb* but does not add the component dimension. This prevents us from observing and studying the impact of WAN links on a per-component basis. To allow catching any correlation between the latency and the performance of the component whether it is deployed on the master site or some workers, we added the component dimension.

It is also worth it to mention that using Prometheus was advantageous in the sense that we were able to setup a post-mortem analysis pipeline using VictoriaMetrics [9] snapshots. Prior to running the experiment, VictoriaMetrics is deployed on the cluster then the Prometheus configuration file is patched and a write backend is added which instructs Prometheus to also forward writes to VictoriaMetrics for long-term storage, snapshoting and post-mortem analysis.

Finally, we highlight that all experiments have been executed on Grid'5000 (all the nodes have been allocated from the same cluster: Intel Xeon Gold 6130, 192 GiB RAM, 1T HDD, 100 Gbps Omni-Path. Experiments have been repeated multiple times to avoid any biases. Artifact and results are available at https://gitlab.inria.fr/kmanaoui/k8sdc

[3] A YAML file that describes the object with variable attribute values that will be assigned by a program (ClusterLoader) at the moment of submitting the object to the api-server.

3.2 Pod Startup Analysis

Pods are a central object in Kubernetes, they represent the containerized workload to be run on the cluster. Thus analyzing the effects of latency on creating, scheduling and running the Pods (Pod startup) is crucial in such a setup. Using the aforementioned artifact, we measured Pod startup time with and without the presence of latency between the master and the worker nodes. For each test, we, iteratively, set the latency to one of the following values: the baseline (. at a LAN scale), 50 ms, 250 ms or 400 ms. We chose these values because they are representative to possible inter data-center latencies [23].

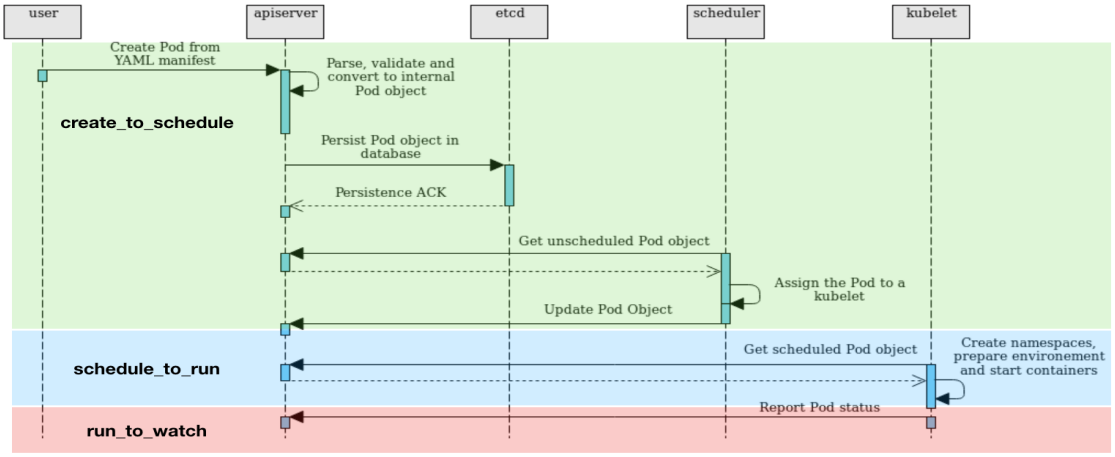


Figure 3: Pod startup sequence diagram

Figure 3 gives the complete sequence diagram of a Pod creation. It can be divided into three main phases:

- **create_to_schedule:** A Pod is scheduled when the scheduler updates the Pod object and assigns it to a worker. Following the assignment, the scheduler updates the Pod status and injects the scheduling timestamp.
- **schedule_to_run** The Pod becomes running when the Kubelet notices the assignment, launches the Pod on the worker and the processes within the Pod starts running. A subsequent status update is made by the Kubelet to reflect the running state with the corresponding timestamp.
- **run_to_watch** The Pod is watchable at the moment the `kube-apiserver` returns the same Pod object in the running state in response to an HTTP/LIST or an HTTP/GET request.

A measurement probe is implemented in ClusterLoader to gather the different timestamps by reading Pod status objects from the `kube-apiserver` and building an internal phase-transition map. Breaking the latency analysis into phases gave us more insights on the the functions and the components that were affected by the WANWide latency.

The results are reported in Figure 4 as the 90th percentile of the three main phases of the creation process. We can see the effects of WANWide latency on the Pod startup times. The transition from the created phase to the scheduled phase is relatively the same for every configuration and we can observe that it is not affected by the WANWide latency. This can be

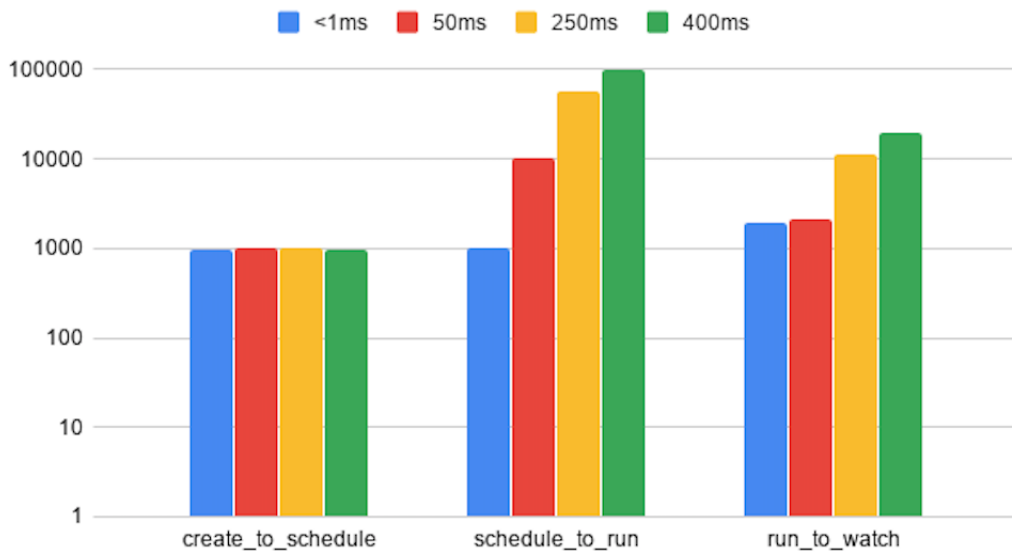


Figure 4: Pod startup latency (in milliseconds)

explained by the fact that no WANWide communication is involved in such a transition and all the actions are taken by components that reside on the master node, namely, the apiserver, etcd and the scheduler. However, the upcoming phases are affected by latency and whenever it is increased, the time required to make the transitions becomes larger. Nevertheless, the `schedule_to_run` transition is slightly more affected than the `run_to_watch` transition. To explain that, it is important to know that the latter encompasses only the status event generation that marks the transition and the event delivery delay which is directly affected by the latency. In contrast, the former encapsulates much more complex logic required to setup the namespaces and prepare the containers file systems including calling the underlying container runtime manager(Docker) and fetching any necessary objects from the apiserver to contextualize the containers (setup configuration files from fetched *ConfigMaps*, create certificates and keys from *Secrets*, etc.). For the sake of clarity, we do not illustrate those possible rounds of communication with the apiserver on the sequence diagram but it is important to understand that each one of them is affected by the WANWide latency.

To verify that the degradation is only due to the communication delay and it is not related to a loss of performance in the control plane nor because of other unexpected errors such as timeouts, we measured the API requests latency. Figure 5 shows the results we obtained. The graph on the top shows the duration of the API requests made from the kubelets and kube-proxies while the bottom one shows the duration of the requests made from the master based components. The graphs reveal the trends we were expecting: master-based requests are not affected by any kind of behavior that may relate to the latency, while the requests issued from the workers take at least, but not so much from, the round trip time duration (that is twice the emulated latency). This proves that the performance is solely affected by the communication delay.

It is worth to note that we did not also encounter any kind of errors, timeouts or retries during the experimentation campaign, and even when injecting up to 10% of packet loss [4] We

[4]Due to space issue and for the sake of clarity these graphs are not presented in this paper but can be generated

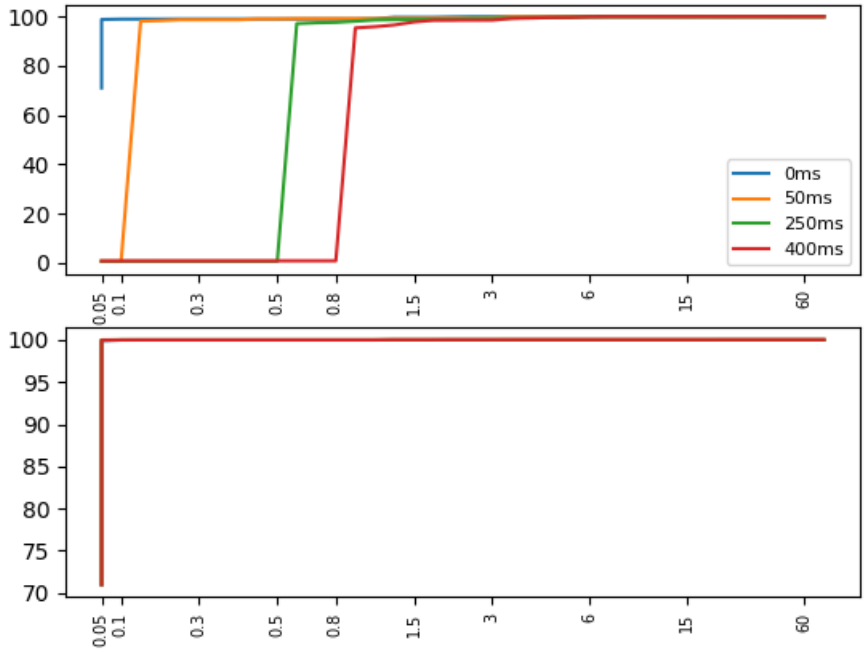


Figure 5: Cumulative Distribution Form of API requests latency reported in seconds and percentages. The graph on top represents the CDF of requests issued from the kubelets and kube-proxies (worker-based components). The graph on the bottom represents the requests issued from master-based ones. Since master based communications are not impacted by the latency, all curves overlap

attribute the reliability of Kubernetes in such a context to its design and more precisely its highly reliable HTTP server and clients, as well as the workers pulling model. More generally, the pulling model for the workers sounds to be a good approach to deal with transient network failures as it will enable workers to quickly recover and converge to its expected state. If the impact on Pods creation is not critical, such a centralized control plane could be considered at first sight.

3.3 Service Discovery Analysis

The goal of the second series of experiments was to evaluate how high level services generally used to deal with the distribution of resources are impacted by the latency. In this section, we present the results we obtained for the service discovery of Kubernetes. A fundamental principle in Kubernetes design is abstracting the underlying system topology and relieving the developer from the burden of dealing with the infrastructure. This principle enables flexible changes in the system as well as the software in an independent manner.Service discovery is a key element to achieve this goal. In Kubernetes, "the failure is the norm": hosts and network can break at any moment, and thus, no assumption should be made regarding Pods locality (they can be rescheduled to other nodes to deal with failures). As a result, it should not be assumed they have constant IP addresses. To let applications communicate without having to keep track of each other's IP addresses, Kubernetes relies on a DNS service through the Kubernetes Service objects

directly from the data-sets available on the git repository.

(service discovery). A Service object is the way by which a Pod or a group of Pod replicas can be made reachable using a service name. It references all the Pods that share the same labels. When a Service object is created by a DevOps, the service controller will assign it a virtual IP address. A Kubernetes DNS server (`kube-dns`), running as a ReplicaSet on the cluster, will notice the new object and reacts by creating DNS records that map the Service name to its assigned virtual IP address.

In parallel, the `kube-proxy` on every worker node, will also notice the new Service object. They will extract its virtual IP address and fetch the IP addresses of all the Pods that it references, using the label selector. They finally alter the kernel's networking stack (through `iptables` or `IPVS`[5]) so that any request destined to the Service's virtual IP will be forwarded to one of the Pods referenced by this Service.

With the same deployment topology shown in Figure 2, we ran a set of experiments to observe the effect of latency on the DNS resolution process and by transitivity on HTTP requests. To this aim, we deployed an HTTP Nginx server and exposed it with a Service object. In a different Pod, we independently sent DNS requests to resolve the Nginx's service name followed by an HTTP request to the Nginx Pod using the same name. Because, the Pods do not contain any processes that cache DNS records, this second HTTP request will imply another DNS resolution request as part of the HTTP function call. In every experiment, we execute the requests in a loop and we log the time it took to return from each function call that performs each type of requests.

For the baseline (at LAN scale), we observed an average latency of 0.693ms and a standard deviation of 0.218ms. For the HTTP requests, the average resolution time was 1.513ms with a standard deviation of 0.240ms. Since most of the languages frameworks and libraries implement connection pools, we expect the HTTP requests time to drop to less than a second which perfectly meets our expectations of the HTTP performance in such a LAN context. With 50ms latency, we observed two distinct collections of request latencies. In the first, values were comparable to the baseline ones. In the second, the duration grows up to an average of 100ms for DNS requests and 101ms for HTTP ones (with both having negligible standard deviations). To explain this difference between these two collections, we must refer to the cluster's architecture in detail. As it was previously cited, Kubernetes relies on `kube-dns` to provide named services.This service is deployed as any Pod through a ReplicaSet with a factor of two (Kubernetes should ensure that there are two Pods at any time). The deployment script we are using in our artifact (`kubespray`) leads to a situation where the first `kube-dns` Pod is always deployed on the master (the second one can be scheduled at any worker).

Like any other Pods, the `kube-dns` service is exposed through a virtual IP address and each worker uses `IPVS` to load-balance the DNS requests among the two replicas with a round-robin load-balancing strategy. With this in mind, it becomes obvious, that some requests are handled by the `kube-dns` replica running on the master site whereas the other ones are satisfied by the second Pod that is deployed in the same location of our client (the Pod that performs the DNS and HTTP requests). This explains the two respective collections of latencies we observed.

Even though the client and the Nginx Pod are deployed on the same location, they might suffer from latency penalties due to the DNS resolution service. This behavior is critical and must be taken into account in multiple site deployment by enforcing the deployment of one `kube-dns` Pod per site as well as the IPVS load-balancing strategy and the nodes of that region should be configured to prioritize dispatching DNS requests to that replica. Technically speaking, IPVS load-balancing strategy could be configured to use the shortest expected delay algorithm or a weighted round-robin.

[5]IPVS is a transport-layer Load-Balancing facility in the Linux kernel. It is the mechanism behind the Services implementation on Linux.

This second series of experiments showed us that using a single instance of Kubernetes to operate an Edge geo-distributed infrastructure must be achieved with care as other side-effects and unexpected behaviors might appear. These observations argue in favor of new kubernetes-based proposals.

4 Related work

As Kubernetes becomes trendy, a few academic [18, 22, 20] and open-source initiatives [26, 4, 8] have been investigating its usage in multi-cloud and edge setups. In the following paragraphs, we present the state of the art of the proposals that are still active, and we believe, the most promising.

4.1 KubeEdge

KubeEdge [26] is an open-source CNCF project that extends Kubernetes under two perspectives: geo-distribution aspects and edge device management. In this section we describe changes related to the first part. Dealing in detail with the second part is out of the scope of our article. However, one can mention that KubeEdge integrates native device management capabilities to Kubernetes and seamless integration with MQTT as well as a standardized interface to discover and query edge devices from within the containers.

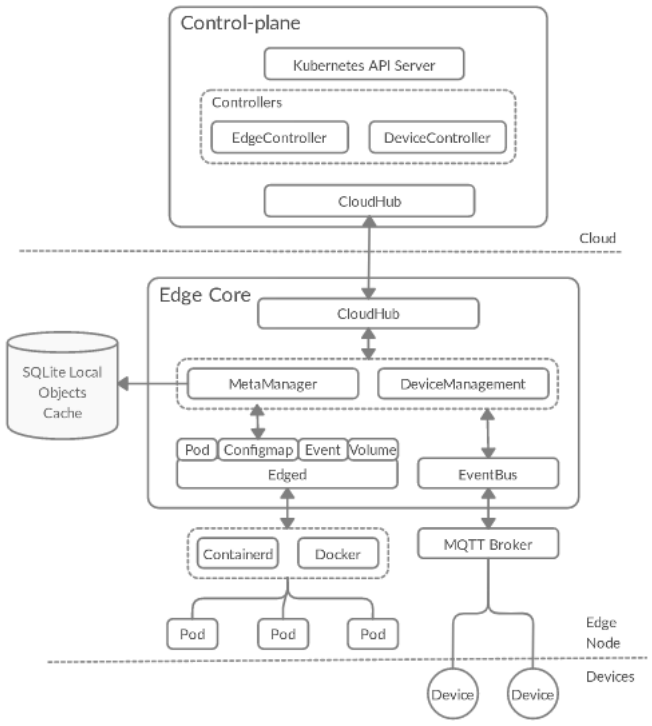


Figure 6: KubeEdge Architecture consisting of a control-plane and an edge node (a worker). EdgeCore represents the Kubernetes components on the edge node

From the architecture viewpoint, KubeEdge represents a significant departure from the vanilla

Kubernetes as depicted by Figure 6. New elements to deal with the the nature of WAN links have been added. First, it adds caching capabilities to edge-nodes using a SQLite database. With such local caching, communication overhead is reduced between the nodes and the master. For example, a lot of the metadata, such as ConfigMaps and Secrets, can be directly fetched from the local cache instead of resorting to the distant master. Another major design aspect in KubeEdge is the rethinking of the communication model between the master and the edge-nodes. In vanilla Kubernetes, the master communicates with the Kubelets by the means of RESTful HTTP connections. In KubeEdge, communication is message-based using WebSockets that provide full-duplex, asynchronous communication channels over a single TCP connection. This model enables greater control of the communication strategy between the master and the edge-nodes and the overhead is further decreased as messages can be queued and then asynchronously sent in a periodic fashion. To support the previously mentioned architectural changes, new components have been introduced to KubeEdge. On the master side, a device-controller is added to support the control of edge devices as well as reporting their status. The edge-controller is an extension of the vanilla Kubernetes controller that sits as a bridge between the kube-apiserver and the edge-nodes, providing event channels and orchestrating state synchronization. CloudHub and EdgeHub are, respectively, the components that implement message-based asynchronous communication over WebSockets on the master and the edge-nodes. Edged is an extension of the Kubelet that supports device management and other KubeEdge facilities. The meta-manager sits between the message layer and edged and implements local caching. When objects arrive from the kube-apiserver, meta-manager stores or updates them on the local SQLite database and, similarly, when the Pods request objects, such as ConfigMaps and Secrets, meta-manager fetches them from the local cache, if they already exist, to avoid communication with the distant master. Otherwise, a request is sent to the master.

Although KubeEdge represents an ambitious project, it does not exhibit any fundamental changes. In addition to limitations previously discussed for the vanilla WANWide Kubernetes (such as the DNS placement problem), the KubeEdge proposal is still based on a centralized control-plane approach. Due to isolation risks of an edge site from the rest of the infrastructure, this model has important limitations. If the master site cannot be reached, it is impossible to provision nor reconfigure workloads hosted on reachable workers at the edge (lack of autonomy). A federated approach presents a significant advantage in comparison to the centralized one: each site can continue to operate locally in an independent manner and communicates with others only if needed. We present in the next paragraph two projects that aims to deliver such a global vision as the vanilla Kubernetes does not provide any mechanism to deliver such a federation.

4.2 Kubernetes Federation

A notable contribution that partially provides cluster autonomy is driven by the multi-cluster working group of the Kubernetes open-source community. The current version is called Kubefed [4], which stands for Kubernetes Federation. The development of Kubefed was driven by the desire of managing in an unified way the life cycle of a multi-cluster workload (a micro-service that should be deployed in multiple regions). Architecturally, Kubefed is a centralized server that distributes and propagates Kubernetes API objects to multiple clusters. Figure 7 illustrates the design.

Kubefed is implemented as an extension of the Kubernetes API by leveraging the use of CustomResourceDefinitions [6]. Basically, all the API resources (`Pods`, `Deployments`, `Services`, etc.) that need to be managed by the federation control-plane, must be declared as a new augmented type using a `FederatedTypeConfig` object. The federated type object (`FederatedDeployment`)

[6] CustomResourceDefinitions (CRDs) are a mechanism to provide user-defined data types in Kubernetes.

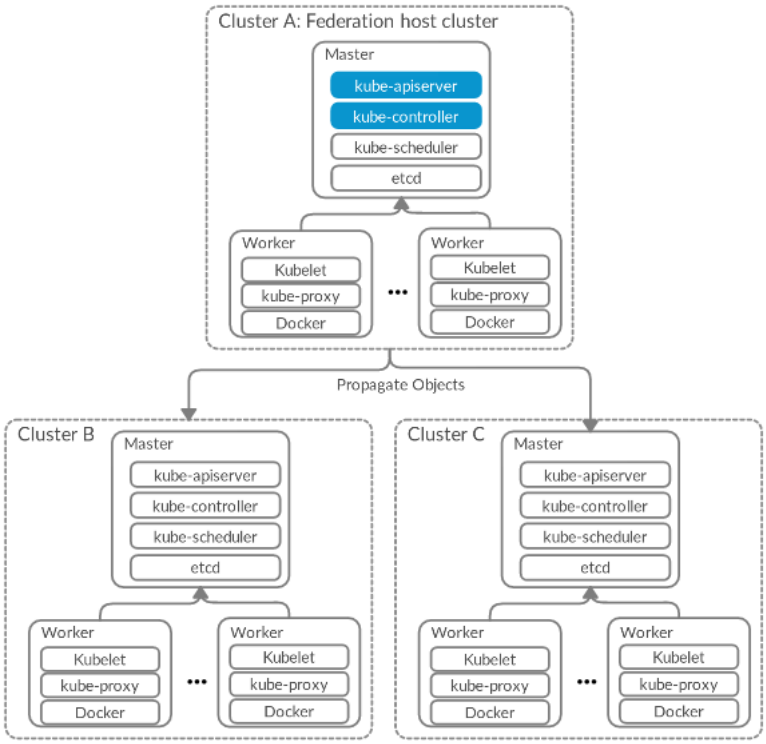


Figure 7: Kubefed Architecture. The master components in blue in the federation's host cluster are respectively the federation's apiserver and the federation's controller manager.

references a template of the underlying API object (`Deployment`) as well as, a set of federation-specific objects that controls the distribution to the different clusters. Namely speaking, those objects are the `Placement`, the `Overrides` and the `SchedulingPreferences`. The `Placement` object selects the clusters to which the underlying API object should be distributed. The `Overrides` object defines cluster-specific configurations. The `SchedulingPreference` object controls the way Pods should be distributed across the different clusters.

Although Kubefed has been a relatively successful to unify the management of multiple clusters, it presents important limitations for our edge use-case. First, the federation control-plane has been designed in a centralized manner. Similarly to the previous approaches that rely on a master site, the federation host cluster is critical in the Kubefed model. Second, Kubefed does not implement any sort of knowledge or cooperation between the clusters themselves: each member cluster is not aware that it is part of a larger infrastructure composed of multiple sites. This lack of cooperation can lead to important issue where a DevOps can interact directly with one site and alter the object that has been provided at the federation level (each cluster owns an exact copy of the object that was submitted on the federation control-plane and there is nothing that can prevent divergence). Finally, the approach of building a federated ecosystem on top of the existing one is debatable. In addition of re-implementing a lot of the existing features at the federation level (inter-cluster scheduling), it required to create new object kinds that are not compatible with the general ecosystem. From our point of view, it would be interesting to study how vanilla Kubernetes objects could be extended in order to allow existing Kubernetes-based programs benefit from resources belonging to multiple sites. In the next paragraph, we discuss

one project that goes in such a direction.

4.3 Submariner

Submariner [8] is another open-source project that aims at solving network connectivity between multiple Kubernetes instances. Unlike Kubefed, Submariner enables to expose Pods and Services from one site to another one without requiring a new API. Technically speaking, Sub-

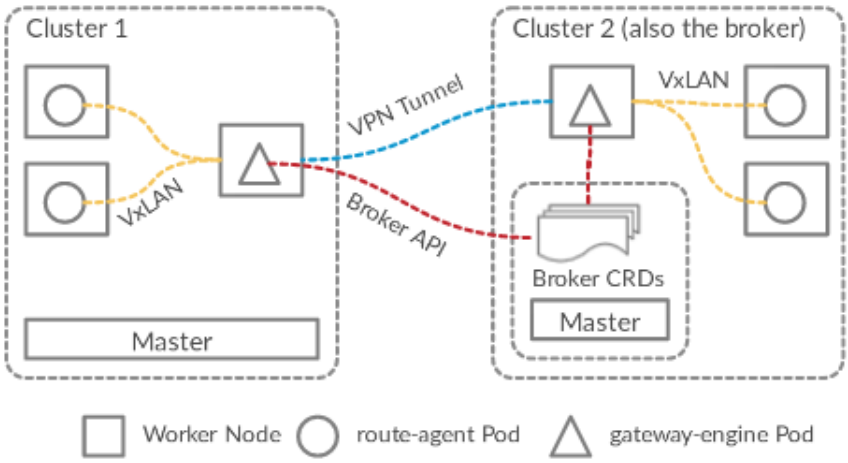


Figure 8: Submariner's Multi-Cluster Architecture

mariner relies on a few internal CRDs which holds essential metadata to establish the inter-cluster communication. The CRDs are, respectively, the `clusters.submariner.io` and the `endpoints.submariner.io`. The former holds metadata about each Submariner cluster: The Pods and Services Classless Inter-Domain Routings (CIDRs), and its DNS suffix. This information is used by each cluster to set up the network routes and the DNS records to reach the Pods and the Services of each other cluster. The latter holds the IP address of the gateway server that is responsible of the inter-cluster routing on each cluster. The CRD objects are all stored in a shared cluster called a broker (usually, just one of the Submariner clusters). All the clusters must be able to contact the api-server of the broker to correctly synchronize their state and share their information. The inter-cluster communication is handled by two major components called the `submariner-gateway` and the `submariner-route-agent` 8. The former ensures inter-cluster communications: on each cluster there is a gateway that peers with its remote twins using IPSec tunnels. The latter is executed on each worker node and is in charge of maintaining the networking rules to correctly route Pod traffic to the remote clusters.

Concretely, when a Pod wants to reach a remote service, the DNS resolution will return its associated ClusterIP. Then, the `iptables` rules will forward the packets to the nodes's route-agent that will encapsulate them and send them to the gateway node by which they will transit and reach the destination cluster. Once they are on the remote cluster, they will be routed depending on the packets destination IP address CIDR. If it is the Pods one, the packets will be routed using the Pods container networking interface (Flannel, Calico...etc). Otherwise, they will be routed using the IPVS facilities as described in Section 3.3.

Submariner has succeeded in solving the challenge of inter-connecting the Services and the Pods of independent clusters in a transparent manner. Applications hosted on distinct clusters

can reach each other using the native Kubernetes service names, overlay networks and DNS servers without having to use any public IP addresses or public DNS servers.Although, Submariner is limited to the networking aspects, it emphasizes the added value of sharing information across multiple clusters. The question now would be to study whether this can be generalized to other Kubernetes objects such as Deployments, Namespaces, etc. so that objects created on one cluster can be exposed to other ones. Nevertheless, the scalability and robustness of such a sharing will be decisive as it will be impossible to maintain a global knowledge base of all created objects. In other words, locally created objects should remain local as much as possible, and only shared with other instances if needed [24].

5 Conclusion

In this paper, we conducted an analysis of the Kubernetes ecosystem under the perspective of an edge computing infrastructure (a cloud computing platform distributed across multiple clusters deployed at the edge of the network).

In the first part, we evaluated the vanilla code in a geo-distributed context. Thanks to its REST-based design, its polling model and the way it manages its states, Kubernetes can manage Pods at a WAN scale without critical issues. However, looking into detail, we observed important side-effects that can affect the managed workloads. We discussed in particular how the placement of general services such as the discovery one is critical to not penalize workloads' performance. It is noteworthy that these experiments have been achieved under specific conditions that should be taken into account. For instance, we did no evaluate the impact of WAN latencies on the retrieval of container images (the image we used was available on each worker) nor whether it is possible to mount a remote volume in a Pod. Evaluating these mechanisms is important as they are also vital for Kubernetes. However, this action goes beyond the goal of this study since they are independent from Kubernetes itself.

In the second part, we described three initiatives that aim at revising the vanilla code of Kubernetes to better deal with the geo-distribution aspects, namely KubeEdge, Kubefed and Submariner. KubeEdge does not exhibit fundamental changes from the software architecture viewpoint as is it still based on a centralized control plane model. Actually only a federated approach such as the one targeted by the two other projects, allows one site to stay fully operational in case of network partitions. However, neither Kubefed nor Submariner succeed to deliver the illusion of a single Kubernetes. By proposing to synchronize network states, Submariner is probably the most interesting project. However, the way to scale such a "sharing" model for all kind of objects is a challenge itself. Moreover, sharing states is not sufficient to deal with the "distribution" aspect. Questions such as how to define a multi-site deployment without building a new API? or How to manage the different reconciliation loops to maintain a global objective are example of challenges that must be addressed by our community.

References

[1] *Kubernetes Scheduler (Documentation)*, 2018 (accessed July, 2020). https://kubernetes.io/docs/concepts/scheduling-eviction/kube-scheduler/.

[2] *ClusterLoader2 Kubernetes test framework*, 2020. https://github.com/kubernetes/perf-tests/tree/master/clusterloader2.

[3] *Etcd database (Documentation)*, 2020. https://etcd.io/docs/v3.4.0/rfc/v3api/.

[4] *Kubernetes Federation Project*, 2020. https://github.com/kubernetes-sigs/kubefed.

[5] *Kubespray, Kubernetes deployment tool*, 2020. https://github.com/kubernetes-sigs/kubespray.

[6] *Performing RollingUpdates on Kubernetes*, 2020. https://kubernetes.io/docs/tutorials/kubernetes-basics/update/update-intro/.

[7] *The Prometheus distributed monitoring framework*, 2020. https://prometheus.io/docs/introduction/overview/.

[8] *Submariner, connected Kubernetes overlay networks*, 2020. https://github.com/submariner-io/submariner.

[9] *VictoriaMetrics timeseries database*, 2020. https://github.com/VictoriaMetrics/VictoriaMetrics.

[10] *Airship*, (accessed July, 2020). https://www.airshipit.org.

[11] *The Cluster API Book*, (accessed July, 2020). https://cluster-api.sigs.k8s.io.

[12] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, et al. Adding virtualization capabilities to the grid'5000 testbed. In *International Conference on Cloud Computing and Services Science*, pages 3–20. Springer, 2012.

[13] Burns Brendan, Grant Brian, Oppenheimer David, Brewer Eric, and Wilkes John. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[14] Justin Cappos, Matthew Hemmings, Rick McGeer, Albert Rafetseder, and Glenn Ricart. Edgenet: a global cloud that spreads by local action. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 359–360. IEEE, 2018.

[15] Ronan-Alexandre Cherrueau, Adrien Lebre, Dimitri Pertin, Fetahi Wuhib, and João Monteiro Soares. Edge computing resource management system: a critical building block! initiating the debate via openstack. In {*USENIX*} *Workshop on Hot Topics in Edge Computing (HotEdge 18)*, 2018.

[16] Ronan-Alexandre Cherrueau, Dimitri Pertin, Anthony Simonet, Adrien Lebre, and Matthieu Simonin. Toward a holistic framework for conducting scientific evaluations of openstack. In *2017 17th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, pages 544–548. IEEE, 2017.

[17] The OpenStack Foundation. *Edge Computing: Next Steps in Architecture, Design and Testing*, June 2020, (accessed July, 2020). https://www.openstack.org/edge-computing/edge-computing-next-steps-in-architecture-design-and-testing.

[18] Tom Goethals, Filip De Turck, and Bruno Volckaert. Fledge: Kubernetes compatible container orchestration on low-resource edge devices. In *International Conference on Internet of Vehicles*, pages 174–189. Springer, 2019.

[19] Stephen Hemminger et al. Network emulation with netem. In *Linux conf au*, pages 18–23, 2005.

[20] Endah Kristiani, Chao-Tung Yang, Yuan Ting Wang, and Chin-Yin Huang. Implementation of an edge computing architecture using openstack and kubernetes. In *International Conference on Information Science and Applications*, pages 675–685. Springer, 2018.

[21] Container Orchestration, Steve Buchanan, Janaka Rangama, and Ned Bellavance. Introducing azure kubernetes service, 2019.

[22] Claus Pahl and Brian Lee. Containers and clusters for edge cloud architectures–a technology review. In *2015 3rd international conference on future internet of things and cloud*, pages 379–386. IEEE, 2015.

[23] Valerio Persico, Alessio Botta, Antonio Montieri, and Antonio Pescapè. A first look at public-cloud inter-datacenter network performance. pages 1–7, 12 2016.

[24] David Sarmiento, Adrien Lebre, Lucas Nussbaum, and Abdelhadi Chari. Multi-site connectivity for edge infrastructures diminet: Distributed module for inter-site networking. In *The 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing*, 2020.

[25] Deepak Vohra. In *Kubernetes Management Design Patterns*, pages 49–87. Springer, 2017.

[26] Ying Xiong, Yulin Sun, Li Xing, and Ying Huang. Extend cloud to edge with kubeedge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*, pages 373–377. IEEE, 2018.

Contents

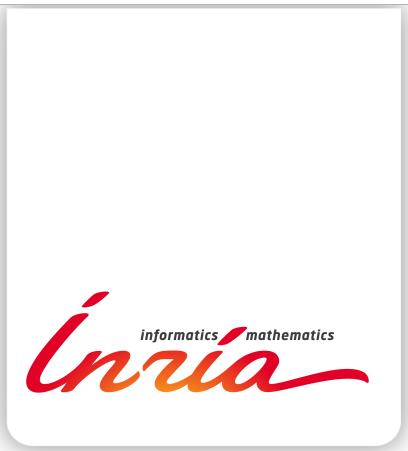
1 Introduction **3**

2 Kubernetes **4**
2.1 Kubernetes Design Principles 4
2.2 Kubernetes Objects 4
2.3 Kubernetes Architecture 5

3 WANWide Kubernetes **7**
3.1 Experimental protocol 7
3.2 Pod Startup Analysis 9
3.3 Service Discovery Analysis 11

4 Related work **13**
4.1 KubeEdge 13
4.2 Kubernetes Federation 14
4.3 Submariner 16

5 Conclusion **17**



RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399