This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

Extending Kubernetes Clusters to Low-resource Edge Devices using Virtual Kubelets

Tom Goethals, Filip De Turck, and Bruno Volckaert

Abstract—In recent years, containers have gained popularity as a lightweight virtualization technology. This rise in popularity has gone hand in hand with the adoption of microservice architectures, mostly thanks to the scalable, ethereal and isolated nature of containers. More recently, edge devices have become powerful enough to be able to run containerized microservices, while remaining flexible enough in terms of size and power to be deployed almost anywhere. This has triggered research into several container placement strategies involving edge networks, leading to concepts such as osmotic computing. While these container placement strategies are optimal in terms of workload placement, current container orchestrators are often not suitable for running on edge devices due to their high resource requirements. In this article, FLEDGE is presented as a Kubernetes-compatible container orchestrator based on Virtual Kubelets, aimed primarily at container orchestration on low-resource edge devices. Several aspects of low-resource container orchestration are examined, such as the choice of container runtime and how to realize container networking. A number of evaluations are performed to determine how FLEDGE compares to Kubernetes and K3S in terms of resource requirements, showing that it needs around 60MiB memory and 78MiB storage to run on a Raspberry Pi 3, including all dependencies, which is significantly less than both studied alternatives.

Index Terms—Edge networks, edge computing, container orchestration, containers, VPN

1 INTRODUCTION

In recent years, containers have quickly gained popularity as a lightweight virtualization technology, owing to their limited resource requirements and fast spin-up times compared to virtual machines [1]. In the cloud, the rise of containers has gone hand in hand with the adoption of microservice architectures, mostly thanks to their isolated and ethereal nature [2]. The complexity of managing large amounts of containers has led to container orchestrators such as Kubernetes [3] which manage the lifecycle, scaling and load-balancing of containerized service deployments.

More recently, edge devices have become powerful enough to be able to run containerized microservices, while remaining flexible enough in terms of size and power consumption to be deployed almost anywhere. This has triggered a wave of research and development aimed at deploying containers on clusters including edge devices, and moving containerized workloads from the edge to the cloud and vice versa. This trend, starting with cloud offloading and edge offloading, has led to concepts such as osmotic computing [4], which aims to optimally distribute workloads based on any number of geographical, hardware and software parameters.

Such a strategy is useful in IoT data processing, where instead of sending large amounts of raw data directly to the cloud, containers are deployed on edge devices to preprocess the data and act on it locally. Aggregated, filtered data can then be sent to the cloud for reporting, advanced processing or to fine-tune the system.

Other examples exist in the field of machine learning, where dynamically moving basic algorithms to edge devices can cut response times drastically while reducing traffic and the load on the cloud. On the other hand, edge devices have to be powerful enough to actually run these algorithms, meaning that constant monitoring is needed in case the cloud needs to step in when the workload is too high.

The applications stated above need container orchestrators that can work optimally in the cloud and on the edge, but most container orchestrators are meant to run in the cloud. As a result, they are usually very flexible, modular and do not need to be overly critical of resource consumption. However, edge devices are typically low-resource devices, especially in terms of memory. Additionally, container deployments on edge containers are often specifically meant for a particular device and cannot easily be relocated without an extensive migration process. To address this, the proposed solution is aimed at minimal resource use, and is designed to run workloads to completion instead of constantly scaling and shifting them.

In addition to being low-resource devices, consumer-grade edge devices often operate in heterogeneous networks with potentially less focus on organization and security. Connecting these to the cloud can result in some communication and security problems. For example, the network could be hidden behind a router, IP addresses can be unpredictable, existing port mappings can interfere with container requirements, etc.

In the cloud, these problems are usually not present. Infrastructure is well-organized and as homogeneous as possible, while all network resources are predictable and controlled. Furthermore, while all communications between container orchestrator nodes (e.g. Kubernetes) are secured by default, this is not always the case for service endpoints of containers deployed on those nodes.

Therefore, it is important that the solution can secure

Manuscript created June 11, 2019.
The authors are with the Internet Technology and Data Science Laboratory, Ghent University - imec, 9052 Gent, Belgium (e-mail: togoetha.goethals@ugent.be, filip.deturck@ugent.be, bruno.volckaert@ugent.be).

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

traffic to and from the cloud, and that other network traffic is either blocked or also secured. To simplify the inter-node network and to ensure smooth deployment of containers, a uniform network environment can be created for edge nodes to be deployed on, capable of supporting a container network on top of it.

Continued development of container management tools such as Kubernetes and Docker [5] has led to the development of a number of standards.

For example, to make sure that every container is reachable and uniquely addressable, container runtimes and orchestrators use an overlay network to assign IP addresses to nodes and containers. Since there are many methods to achieve this on individual nodes, various container runtimes leave this up to network plugins to implement. This has resulted in, among others, the Container Network Interface (CNI [6]), which simply defines a number of high-level operations that governing software can use to organize the container network on a node.

Another example is the Open Container Initiative (OCI [7]), which defines standards for the structure and execution of container images. These standards make sure that a single container image can be deployed and executed with predictable results on any runtime that implements them. At the time of writing, they are implemented by many container runtimes, making it easy to switch runtimes once an orchestrator has basic support for one of them.

The solution is aimed at maximum compatibility with existing container standards, as far as their implementation is possible on edge devices. While it is not absolutely required to implement the full standards, care is be taken that any missing functionality does not result in problems for the rest of the cluster. In addition, if the solution does ignore any standards, it should make sure that other nodes are not affected in any way.

To summarize, the requirements for the proposed container orchestrator for edge devices are:

- To be compatible with modern standards for container orchestration, or to provide an adequate alternative.
- To provide secure communications between edge devices and the cloud by default, with minimal impact on local networks.
- To have low resource requirements, primarily in terms of memory but also in terms of processing power and storage.

This article presents FLEDGE as a low-resource container orchestrator which is capable of directly connecting to Kubernetes clusters by incorporating modified Virtual Kubelets [8] and a VPN. A Kubelet is the part of Kubernetes which is deployed directly on devices to join them into a Kubernetes cluster. A Virtual Kubelet is a small software service which can be deployed anywhere, and which acts as a proxy between the Kubernetes API and a random device that can deploy containers. Behavior is defined by brokers, which form a translation layer between Kubernetes and the devices on which pods are deployed. Because the Virtual Kubelet is designed to work with the Kubernetes orchestrator, it has to work with the limitations inherent in Kubernetes clusters. Kubernetes is designed for use in the cloud, so it is implicitly aimed at orchestration on groups of powerful servers. Since the scalability of Kubernetes constantly evolves, it is considered outside the scope of this article, but the repercussions are discussed where applicable.

The remainder of this paper is structured as follows. Section 2 presents existing research related to the topics in this introduction. Section 3 details the different aspects of using Virtual Kubelets and creating the framework, while Section 4 discusses possible alternatives and how they relate to this work. In Section 5, an evaluation setup and methodology are presented to compare the solution in this article to similar, popular orchestrator software. The results are presented and discussed in Section 6, with suggestions for future work in Section 7. Finally, Section 8 gives a short overview of the goals stated in this introduction, and how the results and conclusions meet them.

2 RELATED WORK

Shifting workloads between the cloud and edge hardware has been extensively researched, with studies on the use of edge offloading [9], cloud offloading [10], [11], fog computing [12] and osmotic computing [4]. Many studies exist on different container placement strategies, from simple but effective resource requests and grants [13], to using deep learning for allocation and real-time adjustments [14].

Kubernetes is capable of forming federations of multiple Kubernetes clusters [15], but this article aims to use a single cluster for both the cloud and the edge. There are several federation research projects that have resulted in useful frameworks, such as Fed4Fire [16], Beacon [17], FedUp! [18] and FUSE [19]. Fed4Fire requires the implementation of an API to integrate devices into a federation and works on a higher, more abstract level than container orchestration. BEACON is focused on cloud federation and security as a function of cloud federation. FedUp! is a cloud federation framework focused on improving the setup time for heterogeneous cloud federations. FUSE is designed to federate private networks in crisis situations, but it is very general and primarily aimed at quickly collectivizing resources, not for deploying specific workloads across edge clusters.

Studies exist that focus on security between the edge and the cloud, for example [20] which identifies possible threats, and [21] which proposes a Software Defined Membrane as a novel security paradigm for all aspects of microservices. However, FLEDGE aims to provide only a basic but universal layer of security, leaving advanced security policy up to individual choice.

VPNs are an old and widely used technology. Recent state of the art studies appear to be non-existent, but older ones are still informative [22]. Some studies deal with the security aspects of a VPN [23], while many others focus on the throughput performance of VPNs [24], [25]. While studies exist on using overlay networks in osmotic computing [26], they deal mostly with container network overlays such as Flannel and Weave [27] which are integrated into Kubernetes. Others present a custom framework, for example Hybrid Fog and Cloud Interconnection Framework [28], which also gives a good overview of the challenges of connecting edge and cloud networks. Xu et al. [29] presents a hardware solution against a number of software and

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

physical attacks for untrusted cloud infrastructure, which could be integrated into edge devices.

A study by Pahl et al. [30] gives a general overview of how to create edge cloud clusters using containers. While FUSE [19] is capable of deploying Kubernetes worker nodes on edge devices, the resulting framework is too resource-intensive for most edge hardware. Cloud4IoT [31] is capable of moving containers between edge networks and the cloud, but it uses edge gateways which indirectly deploy containers on minimalistic edge nodes. K3S [32], which has not yet been the subject of academic studies, is based on the source code of Kubernetes. It achieves lower resource consumption by removing uncommon and legacy features, but it requires its own master nodes to run and can not directly connect to Kubernetes clusters. MicroK8s [33] is another Kubernetes-based solution for edge container orchestration. In addition to having low resource requirements, it is easy to set up, has fast starting times and has built-in GPGPU (General-Purpose computing on Graphics Processing Units) and CUDA support. However, it is aimed at creating smaller clusters for testing, CI/CD and small-scale deployments. KubeEdge [34] is a recent development, aiming to extend Kubernetes to edge clusters. Despite being based on Kubernetes, it also is not directly compatible with Kubernetes master nodes and needs an extra cloud component to function properly. While this article presents a Kubernetes-oriented solution, Docker Swarm has been used for similar purposes in fog computing [35].

Kubernetes CRDs (Custom Resources Definition [36]) can be used to a similar effect as a Virtual Kubelet. CRDs allow IoT devices or resources to be registered in Kubernetes, where they can be assigned workloads through through a custom controller. The main difference with a Virtual Kubelet is that the controller must be hosted on Kubernetes and can control all IoT devices simultaneously. Microsoft uses CRDs for the inverse approach; IoT Edge can integrate with Kubernetes [37] by defining IoT Edge workloads as CRDs, which are converted to pods by the IoT Edge Agent so they can be deployed by the Kubernetes scheduler.

Kubernetes has very limited resource monitoring by default. It keeps a rough overview of the total and required CPUs and memory on each node, but these numbers are unreliable when container orchestration itself takes a significant amount of total resources. Several third party systems have come and gone, such as cAdvisor [38] and Heapster [39], many of which are based on the resource metrics API [40] exposed by Kubelets. There are studies that present their own framework, such as PyMon [41], which is a general container monitoring framework. FLEDGE aims to be compatible with the resource metrics API exposed by Kubelets, so tools such as cAdvisor can monitor them the same way as normal worker nodes.

3 FLEDGE

This section details how the requirements put forth in the introduction are met by FLEDGE, starting with a general overview of what a Virtual Kubelet is and how the solution is based on it.

A Virtual Kubelet acts as a proxy for Kubernetes to any platform or device that can run containers. A Virtual

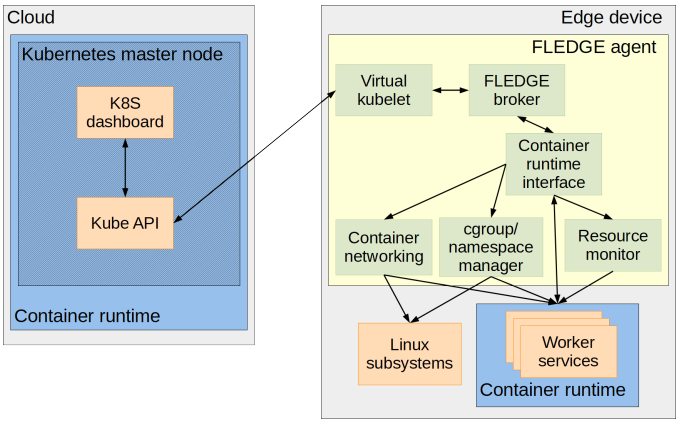


Fig. 1: Conceptual overview of FLEDGE and its use of a Virtual Kubelet.

Kubelet interacts directly with the Kube API on the master node, and passes API calls to brokers that implement them for the system they represent, for example Amazon AWS, Microsoft Azure or an edge device. The API calls supported by a Virtual Kubelet consist of pod management, pod status, node status, logging and metrics.

Fig.1 shows how a Virtual Kubelet fits into the FLEDGE framework. When FLEDGE is started, the Virtual Kubelet is initialized and the FLEDGE broker connects to it, receiving commands from Kubernetes through it. Depending on its configuration, the broker will initialize a specific Container Runtime Interface which decomposes the commands into container networking, cgroup management, namespace management, or passes them on to a container runtime (e.g. Docker, Containerd). The collection of FLEDGE components deployed on an edge device will be referred to as a FLEDGE agent.

3.1 Compatibility

One of the requirements for FLEDGE is that it should support container standards and existing container runtimes. There are a few aspects to this requirement, some of which are limited by the APIs of existing software.

The first aspect is the choice of container runtime. While Docker may seem like a logical choice because it is very widely supported, Containerd is also an option. Since version 1.11, Docker relies on Containerd for some operations, such as container execution. Both runtimes support the OCI standards, so they can both create and run OCI containers (Docker containers). In terms of compatibility, both are valid choices, so ultimately it comes down to a trade-off between ease of implementation and resource requirements, which will be discussed in Section 3.3.

Related to the choice of container runtime is compatibility with Linux cgroups. Some devices and operating systems do not support all cgroups by default, making it hard or impossible to correctly run Kubernetes deployments. On a Raspberry Pi 3 running Raspbian for example, cgroups used for CPU throttling may be missing from the kernel, which must be custom-built in order to guarantee compliance with Kubernetes specs. If these kernel options are missing, FLEDGE will generate a warning, but still continues with

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

the deployment if possible. Since neither Kubernetes nor Docker seem affected by the absence of the cgroups in question, this approach seems to be the standard.

Another aspect of compatibility is how container networking is handled. In Kubernetes, container networking is implemented as an overlay network [42] in which each pod can be assigned a distinct IP address on a virtual network interface. This is achieved by assigning sub-ranges of a configurable IP range to each node, from which they in turn assign IP addresses to their pods. Kubernetes itself makes high-level decisions on container networking, such as assigning IP ranges to the Kubelets on the nodes. The assignment of IP addresses to pods and the setup of network namespaces and virtual network interfaces is handled by CNI compatible network plugins (e.g. Flannel, Weave) on the nodes themselves. To use a specific network plugin, it is deployed on the master node, which in turn makes sure it runs on all worker nodes.

In FLEDGE, this is implemented differently. By fulfilling the role of both Kubelet and container network plugin, there is no need for the CNI layer usually present between Kubelet and network plugin. Additionally, the number of pods that can be deployed on edge devices is rather limited compared to cloud infrastructure. This means that it is preferable to implement a simple and naive, but effective pod networking handler (Fig.1 *Container networking*) which hands out IP addresses on a first come, first serve basis. This pod networking handler is also responsible for configuring networking namespaces correctly (Fig.1 *namespace manager*), independent of the active container runtime. The deployment of the network plugin itself is prevented by labeling the node so it is not eligible for deployment.

Since FLEDGE uses the Kubernetes-assigned IP ranges to configure its container networking, this approach does not influence container networking in the rest of the cluster. The master node is unaware that the node does not deploy the default network plugin and handles its networking needs, and the container networking plugin is still deployed and functioning normally on other nodes.

As stated in the introduction, Kubernetes node resource monitoring is sufficient to determine if any additional pods can be deployed on a node. This monitoring is based on the total resources of a system and the maximum resources allocated to pods, actual resource use is not taken into account. This is a problem for edge devices, since operating system and orchestrator resource use can constitute a significant portion of total resources, making it hard to gauge if a device can take additional load based on pod-allocated resources alone.

Luckily, Kubelets also provide the Resource Metrics API which is used by several third-party monitoring tools. In order to support these monitoring tools, one needs to implement the Resource Metrics API up to a level sufficient for monitoring edge device resources and pods (Fig.1 *Resource monitoring*). By default, the Resource Metrics API is hosted on the same port as on a normal Kubelet.

3.2 Security and stability

Edge devices often find themselves in heterogeneous networks with little to no organization or security. This randomness of topology, IP address assignments and port

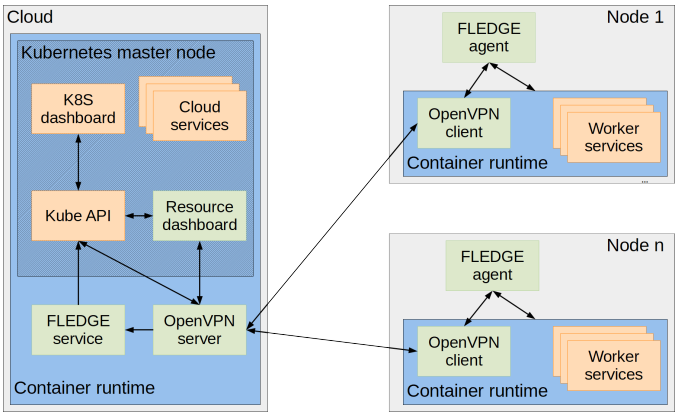


Fig. 2: High-level overview of network traffic flow of FLEDGE, using OpenVPN to connect edge nodes to the cloud.

mappings is not an ideal situation for building a cluster and deploying containers. Furthermore, the situation could be exacerbated by the presence of a router with either NAT or a firewall. Finally, while Kubernetes node traffic is secured by default, the same is not always true of services deployed on nodes, so all traffic between the cloud and the edge should be secured by default.

In FLEDGE, this is solved by setting up a VPN, more specifically OpenVPN, and building the cluster and container network on top of its interfaces. The basic traffic flow of this setup is shown in Fig.2. While simple, this approach fixes the problems described above:

- IP addresses of nodes are predictable and directly reachable by the master node.
- The VPN interface is a proverbial clean slate; all ports are open and available for use.
- Physical layout of the network no longer matters, the VPN can be organized according to logical parameters.
- UDP hole punching might be required to overcome NAT or a firewall, but this is taken into account by OpenVPN.
- Packets are encrypted by default for a basic layer of security.

However, the effectiveness of using a VPN also depends on the software used and its exact configuration:

- The effectiveness of packet encryption depends on the chosen algorithm, and encryption can even be turned off entirely for performance reasons. FLEDGE uses default OpenVPN encryption.
- Using a VPN is a drain on system and network resources, likely reducing the scalability of clusters. OpenVPN has another drawback in that it can only use a single CPU core, which may quickly saturate and limit its performance on edge devices.
- Anyone with physical access to the device can piggyback on the VPN connection and reach any cluster services. Preventing this requires physical and OS-level security.

The custom container network implementation in FLEDGE uses IPtables to configure the routing between

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

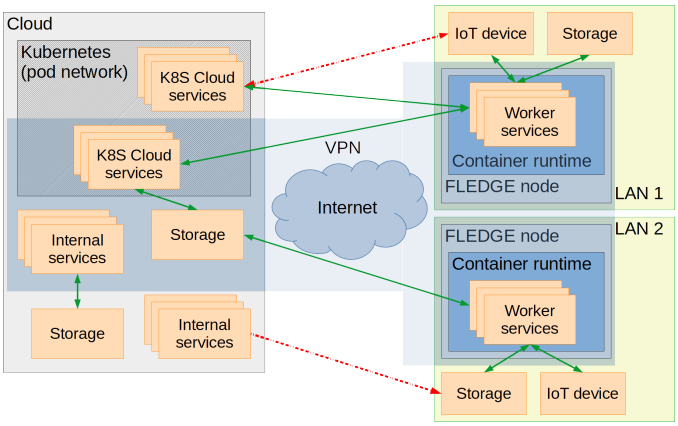


Fig. 3: Overview of network traffic flows in a cluster using FLEDGE nodes. Green arrows indicate possible traffic flows.

pods and the rest of the cluster. Properly configuring IPtables with respect to the VPN interface allows the exclusion of certain traffic flows, as shown in Fig.3. The solid green arrows indicate traffic flows allowed by FLEDGE, showing that any container running on a FLEDGE agent can access any device or container in the VPN and the pod network. Not all devices need to be connected to the VPN, nor do they all need to be part of the pod network. Note however, that traffic from devices in the pod network that are not connected to the VPN can easily be blocked by configuring IPtables differently.

Fig.4 reiterates Fig.2, but on the level of network interfaces. This figure shows how the entire container network is built on the VPN network, and that all traffic uses the VPN interfaces. *CNI* (solid red) arrows represent container network traffic, while *Normal* arrows represent VPN and ethernet traffic. The *Host map* arrows indicate traffic to containers that use the host network namespace, meaning that they have direct access to the network interfaces of the node they run on rather than operating through a virtual network interface. The OpenVPN and FLEDGE containers run in privileged mode, since they need the authority to create and modify network interfaces, cgroups and namespaces. Similarly, they run in the host network namespace because they need to cooperate and create network infrastructure for the container runtime and the master node. While the figure indicates that all container traffic goes via the container network, they can still be assigned to the host network namespace. After all, FLEDGE simply executes Kubernetes deployments. However, putting them on the host network namespace is discouraged, since that might make it harder to communicate with other cluster services.

Container images may contain proprietary software that needs to be protected from local and remote unauthorized access, and the resulting risk of reverse engineering. Because FLEDGE agents have to run as root, they present a prime attack vector to access all of the images and containers they manage. However, a few steps can be taken to mitigate this:

- Running containers are by default assigned to different file system namespaces by most container runtimes. While a root account can easily access the file system of a container, it can be protected against any user that is not root, apart from the user running the container.
- To minimize the chance of images being copied and reverse-engineered, they can be removed when the containers in a pod are finished. While this also frees up some extra storage for reuse, it may slow down redeployments of the same pod because the container runtime needs to download the images again.
- FLEDGE cleans up all network infrastructure, containers and images on shutdown. This is also required for leaving the system in the same state it was in before deploying FLEDGE.

3.3 Low resource use

The choice of container runtime is very important for resource use. Because Docker relies on Containerd to actually run containers, Containerd is likely the most resource-friendly option. On the other hand, the Containerd APIs require more low-level implementation to use effectively than those of Docker.

For low-resource edge devices it is reasonable to put resource requirements before ease of implementation, and since the compatibility section has shown that there is little to no difference in supported standards between the container runtimes, it stands to reason to propose Containerd as the runtime for FLEDGE. The Results section will further validate this choice.

3.3.1 Networking

The compatibility section argued in favor of a custom CNI facility in FLEDGE, rather than using one of the existing containerized network plugins such as Weave or Flannel.

This design choice is optimal in terms of resource requirements, considering that all container plugins are deployed as containers. While using containers is flexible and more durable than other forms of plugins (e.g. host process or in the same process), it also means that a plugin requires a significant amount of resources to run. While this is not explicitly reflected in any results in this article, the requirements for Flannel are determined and discussed in Section 6 to support this claim.

3.3.2 Namespaces and cgroups

Using the low-level APIs of Containerd means that some functionality needs to be implemented explicitly. Two of the most important aspects of this functionality are cgroups and namespace handling.

While both Docker and Containerd create the required namespaces for a new container, FLEDGE takes care of all namespace management after the creation of the first container of a pod. This is to make sure that no matter which container runtime is used, the behavior is the same.

On the other hand, container resource restrictions are much easier to pass directly via the Docker API, which populates the required cgroups automatically. While Containerd is also capable of making cgroups, the actual restrictions need to be set by the program using the Containerd APIs. Therefore, FLEDGE only allows the creation of one cgroup of each type (memory, cpu, ...) per pod. After configuring

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

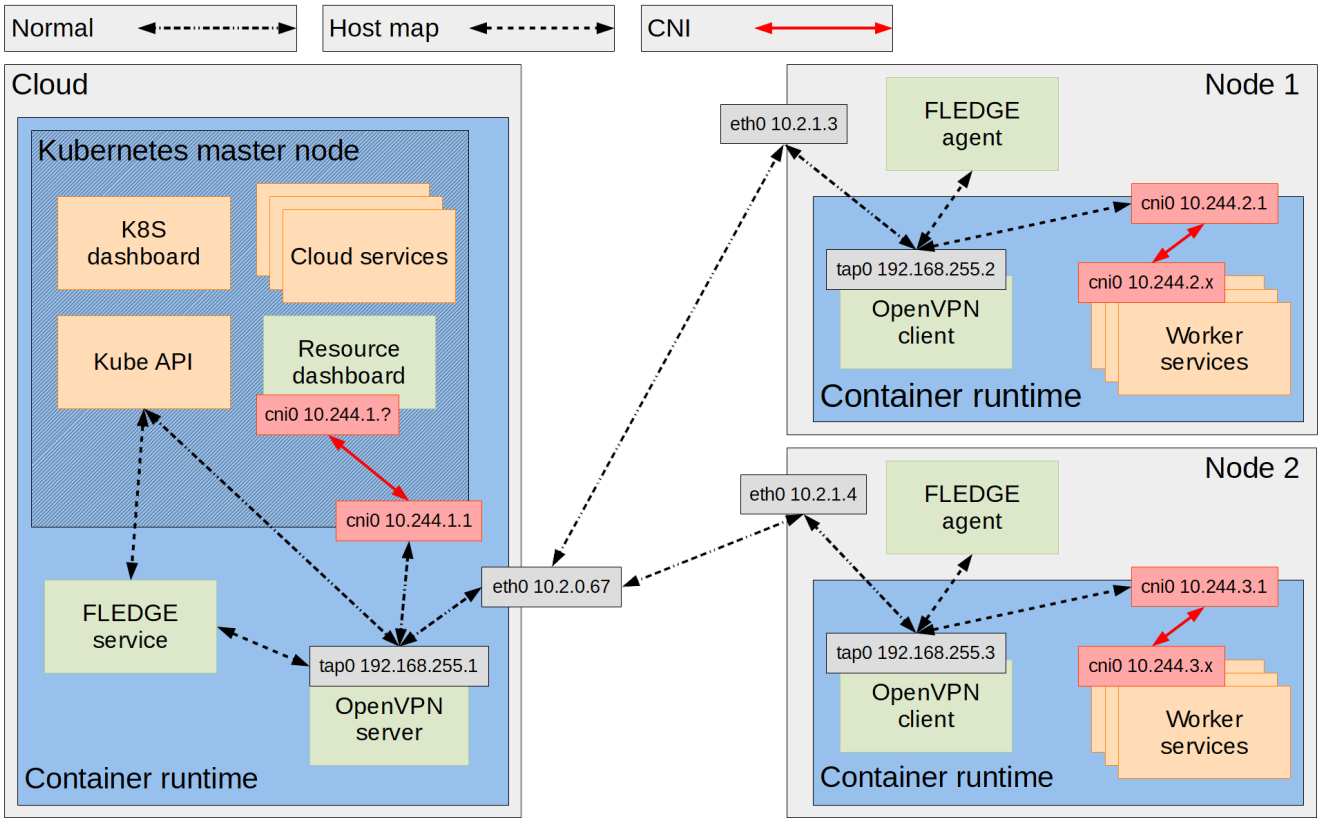


Fig. 4: Overview of network traffic flows in a cluster using FLEDGE nodes, on the interface level. Note that OpenVPN and FLEDGE always run in the host network namespace.

the resource restrictions, it forces Containerd to reuse them for the rest of the containers in a pod.

Similar to container networking, the complexity of cgroup and namespace management on edge devices is much reduced compared to cloud infrastructure. Therefore, despite increasing the complexity of FLEDGE, handling cgroups and namespaces in FLEDGE itself using a minimal implementation allows conserving resources for actual workloads.

3.3.3 *Virtual Kubelet location*

As explained above, the Virtual Kubelet is only a small part of FLEDGE. While instrumental in the communication with Kubernetes master nodes, its location matters very little since a custom broker implementation can forward API calls to other devices.

In terms of resource requirements, this allows for two options when considering where to run the Virtual Kubelet:

- In the cloud: the Virtual Kubelets are run as pods in the cloud, entirely separate from the FLEDGE agents which are running on edge devices. Kubernetes API calls received by Virtual Kubelets are forwarded to FLEDGE agents via REST services. This approach shifts some of the resource requirements from the edge to the cloud, while allowing for a more robust system. For example, when a FLEDGE agent loses its connection to a Virtual Kubelet, the Virtual Kubelet can queue commands and give default responses until the agent comes back online.
- On the edge: the Virtual Kubelet is integrated into the FLEDGE agent and run as a container or a normal process on the edge device. Kubernetes API calls are executed directly in the same process. While this approach requires more resources on the edge and is less resistant against network problems, it does reduce the operational and technical complexity of FLEDGE.

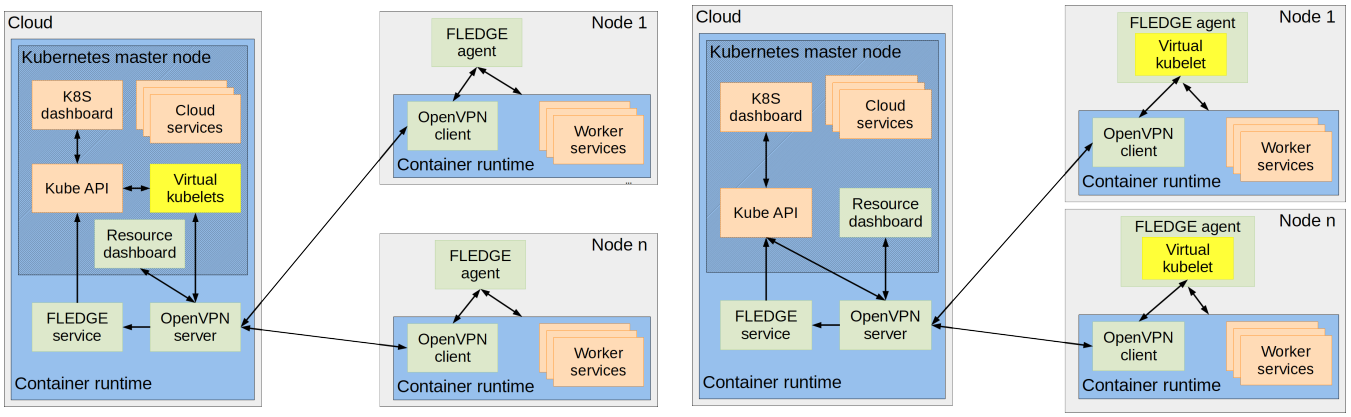
The two options are further illustrated in Fig.5. On the left, the Virtual Kubelets run in the cloud, while on the right the Virtual Kubelet is shown in its pass-through role on the edge device.

Note that when the Virtual Kubelets are run in the cloud, a small web service (FLEDGE service) is required on Kubernetes master nodes to simplify Kubernetes API access for FLEDGE agents. Without this service, FLEDGE agents would have to include the full Kubernetes API, increasing their size by about 20MiB. When the Virtual Kubelet is integrated into the FLEDGE agent, they share the Kubernetes API and the FLEDGE service is no longer required. The resources required for the FLEDGE service are insignificant when deployed on a server, so they will not be taken into account for the rest of the article.

To properly determine where to put the Virtual Kubelet, a model needs to be constructed which takes into account the resource use in both situations, and the relative importance of edge resources versus cloud resources. Kubernetes v1.14 has a limit of 5000 nodes per cluster [43]. Because there is always at least one master node, this means a

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807



(a) Virtual Kubelet in the cloud.

(b) Virtual Kubelet on the edge.

Fig. 5: Overview of FLEDGE architecture for different locations of the Virtual Kubelet. In Fig.5a, the Virtual Kubelet is run in the cloud, communicating with FLEDGE over VPN. Fig.5b shows how the Virtual Kubelet can be integrated into FLEDGE on the edge device, passing Kubernetes calls directly to the FLEDGE broker.

cluster can contain at most 4999 edge nodes. However, if the Virtual Kubelets are deployed in the cloud as pods, the maximum number of pods per node is also important. For v1.14 this limit is 110, but taking plugins and default pods into account, 100 pods per node is a safe estimate. This means that for every 100 edge nodes, there would need to be an additional master node to manage them, increasing the complexity of the management structure in the cloud. Modeling all the requirements starts with calculating the required number of management nodes N_M and management efficiency E:

$$N_M = \lceil \frac{L_N}{L_P + 1} \rceil \tag{1}$$

$$E = \frac{L_N - N_M}{L_N} \tag{2}$$

Where L_N is the limit of nodes per cluster and L_P is the pod limit per node. Eq. (1) and eq. (2) can be used to construct the total memory used by all pods M_{Pods} and nodes M_{Nodes}:

$$M_{Pods} = L_N \cdot E \cdot M_{Pod} + M_{Shr} \cdot (N_M - 1) \tag{3}$$

$$M_{Nodes} = (N_M - 1) \cdot M_{Kube} \tag{4}$$

Where M_{Pod} is the amount of non-shared memory required per Virtual Kubelet, M_{Shr} is the amount of memory shared by all Virtual Kubelets on a node, and M_{Kube} is the amount of memory required for a Kubernetes installation. M_{Kube} can be extended to the memory requirement of an entire operating system or virtual machine, depending on how Kubernetes master nodes are instantiated in cloud infrastructure. Eq. (3) and eq. (4) can in turn be used to calculate the maximum additional amount of memory the Virtual Kubelet should require per edge node for edge placement to be more memory efficient than cloud placement:

$$M_E = C_M \cdot \frac{L_N \cdot E \cdot M_{Pod} + (M_{Shr} + M_{Kube}) \cdot (N_M - 1)}{L_N} \tag{5}$$

Where C_M is a constant representing the relative cost of edge memory versus cloud memory. This constant is important because cloud memory is cheap and easily extensible. Similar to eq. (5), a formula can be constructed for storage requirements:

$$S_E = C_S \cdot \frac{L_N \cdot E \cdot S_{Pod} + (S_{Shr} + S_{Kube}) \cdot (N_M - 1)}{L_N} \tag{6}$$

Where C_S, S_{Pod}, S_{Shr} and S_{Kube} fulfill the roles of C_M, M_{Pod}, M_{Shr} and M_{Kube} respectively. The only factor not considered in these equations is the cost of maintaining a more complex cluster of master nodes in the cloud, which is very case-dependent and hard to estimate.

C_S is assumed to be 1, since the target class of edge devices can routinely store several gigabytes, and the size of a discrete Virtual Kubelet is merely 32MiB on x64. For such small amounts, storage is equally cheap in the cloud and on the edge. Furthermore, edge devices that are not equipped with at least 512MiB (compressed) storage are unlikely to be able to run a Linux based operating system with a kernel capable of handling containers, so it is not useful to consider such devices for container deployment. C_M depends on a lot of factors. Most important of all, cloud memory is often priced in terms of GiB-seconds, while edge hardware is a one time purchase but typically non-extensible. Both types of memory have a wide range of pricing constantly in flux, further complicating attempts to calculate C_M. For the rest of this article, it will naively be assumed to equal 1.

4 ALTERNATIVES

The previous section describes how FLEDGE solves most of the requirements put forth in the introduction, specifically secure communications and compatibility with existing standards and APIs. However, proving that FLEDGE resource requirements are lower than those of comparable software requires some experiments.

This section discusses some alternative container orchestrators, giving a short history and possible advantages and disadvantages for each.

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

4.1 Kubernetes

Kubernetes [3] is a very popular container orchestrator originally inspired by Google Borg [44]. Years of Kubernetes development have contributed to several container standards, some of which have been discussed in previous sections. Kubernetes is made to run in the cloud, and while it is very flexible and extensible, it also tends to use too many resources when deployed on edge devices.

It has already been discussed how the most important elements of FLEDGE relate to Kubernetes in previous sections. However, there is still an important difference between FLEDGE and Kubernetes in that the latter requires all swapping to be disabled. This leads to serious performance and stability issues on some edge devices (e.g. Raspberry Pi 3), which are already low on memory after a Kubernetes deployment. FLEDGE does not require swap to be turned off, so all memory subsystems can perform as intended.

4.2 K3S

K3S [45] is a new container orchestrator based on the Kubernetes source code, but modified specifically for edge devices. Version 0.1.0 was released in February 2019, and while v0.5.0 is currently available, the version used for the evaluations is v0.3.0. Unlike FLEDGE, which is only meant to be used on worker nodes, K3S also has its own master nodes.

Where FLEDGE starts out from scratch and works towards Kubernetes compatibility, K3S takes the inverse approach and eliminates unnecessary code and functionality from the full Kubernetes source code. Unlike Kubernetes, it has no choice of container runtime; Containerd is used by default. Similarly, Flannel is integrated for container networking.

While K3S has better support for Kubernetes APIs, not being built from scratch can be a disadvantage for it in terms of resource requirements. Additionally, it has a slightly different cluster join mechanism and a thin wrapper layer which gives it its own shell commands. These changes mean that, for now, K3S worker nodes cannot be used in a Kubernetes cluster, but only in K3S clusters.

4.3 KubeEdge

KubeEdge [46] is a new, early-stage Edge Computing Framework based on Kubernetes and designed specifically for edge networks. It was initially released in December 2018, with the latest version being v0.3.0 as of early May 2019.

KubeEdge is built on open source software, including Kubernetes and Docker, and aims to provide an ecosystem for container orchestration on edge devices. It consists of a cloud part and an edge part [47]. The cloud part communicates with the Kubernetes API in the cloud and has high-level control over edge devices. The edge part takes care of container deployment and provides an infrastructure for storage and event-based communication, the latter being based on MQTT [48].

Since it is much more than just a simple container orchestrator, including it in the evaluations would not result in a fair comparison for KubeEdge. Despite this, it is unlikely to be a very-resource efficient solution because of its use of Docker, a point which will be proven in the Results section.

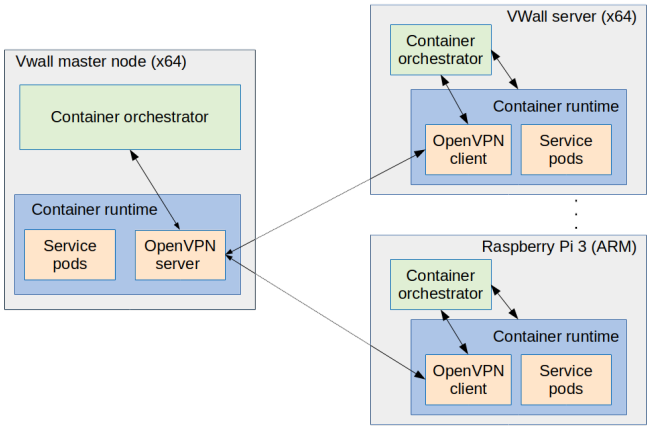


Fig. 6: Overview of the hardware setup used for the evaluations. Note that the OpenVPN containers are only used by FLEDGE, other orchestrators connect directly to the master node via LAN. Previous figures did not show OpenVPN components as containers because they were more conceptual.

5 EVALUATION SETUP

Now that the FLEDGE architecture has been explained and alternative approaches have been identified, an evaluation environment can be constructed. The evaluations are intended to confirm some of the choices made in earlier sections, to back up some claims, and to give an indication of how FLEDGE stacks up against Kubernetes v1.14 and K3S v0.3.0 in terms of resource consumption.

The source code of FLEDGE is made available on Github[1].

This section first gives an overview of the evaluation setup and general methodology. Subsequently, the specifics of each evaluation are explained.

5.1 Methodology

Fig.6 shows the hardware setup used for the evaluations. There are 3 devices involved:

- The *VWall master* node fulfills the role of a Kubernetes/K3S master node. Its specifications are not important, since the worker nodes are the focus of the evaluations.
- The *VWall server (x64)* is used to determine the resource requirements of orchestrator worker nodes on an x64 CPU architecture. This server has an AMD Opteron 2212 processor at 2GHz and 4GiB RAM, running Ubuntu 18.04.
- The *Raspberry Pi 3* is used to determine resource requirements for FLEDGE on an ARM CPU architecture. This device runs Raspbian with kernel version 4.14.98-v7+ on the default hardware configuration; 1GiB RAM and a quad-core 1.2GHz CPU.

All devices are in the same geographical location and are connected by a Gigabit LAN. The OpenVPN server and clients are only used when FLEDGE is deployed on the

1. https://github.com/togoetha/fledge

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

worker nodes, Kubernetes and K3S connect to the master node directly via LAN. All evaluations will be performed on both ARM and x64.

The container runtime used in any evaluation depends on the orchestrator being tested. For Kubernetes, Docker is used, while K3S has Containerd by default. For FLEDGE, both Docker and Containerd are possible.

The storage requirements for each orchestrator are determined by using the df [49] command before and after orchestrator setup. After every evaluation, the devices are wiped to ensure the same state at the start of each evaluation. In addition to the orchestrator and the container runtime, this approach also takes packages and libraries into account that are required to run the orchestrator properly, thus forming a complete picture of storage requirements. Because no deployments or workloads are executed apart from the default containers required for each orchestrator, storage does not vary over time and thus it is not necessary to measure beyond the successful start of each orchestrator.

Measuring memory use is more complex than determining storage requirements, for the following reasons:

- Unlike the thousands of files involved in setting up an orchestrator, the processes involved in running it can be easily identified, so a more granular approach is possible. This is not only more accurate, but allows for more detailed conclusions by studying subsets of processes.
- It stands to reason that memory use is not as static as storage requirements. During deployment, a lot of memory will be used which may be released again later. Therefore, memory use must be monitored over a significant period of time to form a complete picture.
- Processes can have private and shared memory. While it is easy enough to obtain these numbers, a fair method is required to calculate the exact memory use of a process from both numbers.

During each evaluation, memory is measured every 30 seconds over a period of 15 minutes, while the pmap [50] command is used to determine the Proportional Set Size [51] (PSS) of each process, calculated using the following formula:

$$M_{total} = P + \sum^{i} S_i/N_i$$

where P is private memory, S_i are various sets of shared memory, and N_i is the number of processes using any piece of shared memory.

5.2 Container runtime comparison

Previous sections have argued that the choice of container runtime can have a large impact on resource requirements for an orchestrator solution. In order to verify this, FLEDGE is set up as in Fig.5a, using both Docker and Containerd. No pods or containers other than the FLEDGE agent and a VPN client are deployed, to reduce the influence of other processes on memory use behavior. A third case is also examined, in which the FLEDGE agent runs directly on the host while using Containerd as a runtime, to determine the containerization overhead of the FLEDGE agent.

In all cases, the processes monitored are container runtime daemons, the FLEDGE agent, the VPN client and Containerd shims [52].

5.3 Virtual Kubelet integration

As shown in Section 3, Virtual Kubelets can either be deployed on the master node or merged with FLEDGE on edge devices. This evaluation is meant to gather the required data for Eq. 5 and Eq. 6 so an argument can be made for the correct approach.

To gather the required data, FLEDGE is set up as described in both Fig.5a and Fig.5b, and the same processes are monitored as in the Container runtime comparison.

5.4 Orchestrator comparison

As presented in Section 2, there are a number of alternatives to FLEDGE. Since the point of FLEDGE is to provide a Kubernetes-compatible container orchestrator with minimal resource requirements, this evaluation is meant to verify that FLEDGE requires fewer resources than Kubernetes worker nodes on edge devices. To fully prove this, Kubernetes is allowed to deploy a kube-proxy [53] on FLEDGE to level the playing field. Flannel will be used as a CNI plugin, but since FLEDGE has its own container networking it will only be deployed on Kubernetes worker nodes.

Additionally, FLEDGE is compared to K3S to show that it is a useful alternative to K3S. Because K3S does not actually include kube-proxy by default, this evaluation compares K3S to a FLEDGE deployment without kube-proxy.

For this evaluation, the monitored processes are the container orchestrator, the container runtime, shims and any deployed containers (including VPN for FLEDGE). FLEDGE uses Containerd as a container runtime.

6 EVALUATION RESULTS

This section presents the results of the evaluations described in Section 5. For practical purposes, x64 numbers are shown as blue series in the charts, while ARM is shown as red with dashes. Storage requirements charts are bar charts showing medians, memory charts also have error bars indicating the median absolute deviation.

6.1 Container runtime comparison

Fig.7 shows the storage requirements for FLEDGE deployments using either Docker or Containerd.

The first important observation is that on ARM devices, a FLEDGE deployment using either Containerd and Docker requires far less storage than on x64. The difference is especially large in the case of Docker and FLEDGE, which needs 3 times as much storage on x64 as it does on ARM.

At first sight, it appears that Containerd is much less efficient on ARM than Docker is, but this conflicts with the fact that Docker uses Containerd for many container tasks. However, in order to use a containerized version of the FLEDGE agent with Containerd, many files and resources need to be made available inside the FLEDGE agent container for it to be able to deploy containers itself.

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

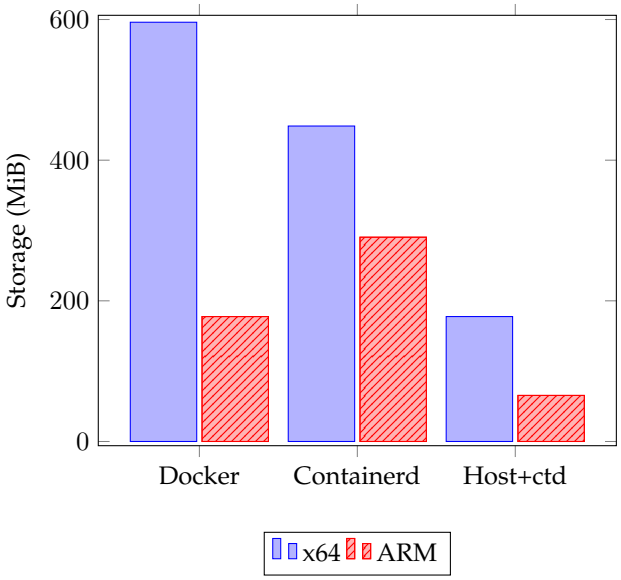


Fig. 7: Storage requirements of FLEDGE using different container runtimes, including all relevant processes. The Host+ctd category shows the results for FLEDGE running directly on the host, using Containerd to run deployments.

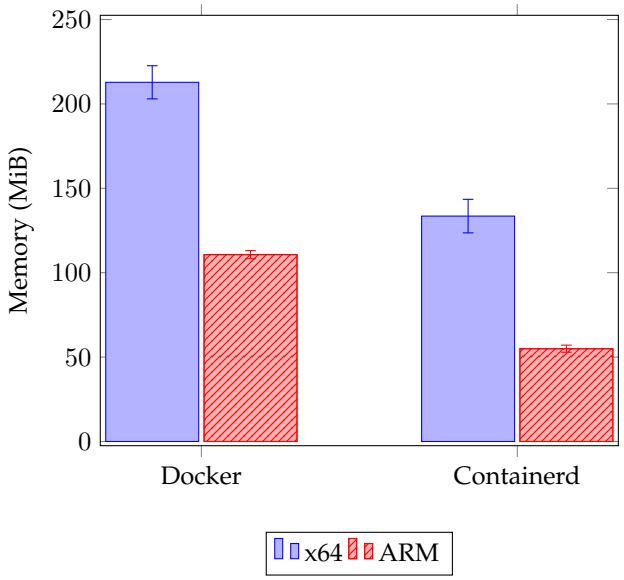


Fig. 8: Memory use of FLEDGE using different container runtimes, including all relevant processes.

It turns out that mounting all these file paths inside the FLEDGE agent container at runtime creates a multitude of file system layers which inflate storage requirements up to 4 times the original size. In order to validate this, a FLEDGE agent was run as a host service with Containerd as a container runtime. Additionally, this version of Containerd was cleaned of unnecessary support executables, most notably the command line tool *ctr*, since only API interaction is required. This is similar to the approach K3S uses, and the most important downside is that the command line can no longer be used for debugging purposes. This approach (Fig.7 *Host+ctd*) is much more resource efficient, using only about one third of the resources Docker requires on both x64 and ARM.

Note that the same approach does not work with Docker; running the FLEDGE agent as a host service with Docker as a container runtime gives nearly the same results as in Fig.7. This indicates that while Docker may use Containerd as a runtime, it has a much more efficient method of creating and mounting file system layers.

The results in Fig.7 can thus be explained by two causes. The first is how mounts are handled by the container runtimes, the second is the result of instruction set differences and larger overall binaries on x64. The effects on required storage are respectively additive and multiplicative. This is reflected in Fig. 7, where the *Host+ctd* and *Docker* categories scale more or less equally between x64 and ARM, but the inflated layers in Containerd are similar added burdens on both x64 and ARM. Note that the differences in the latter case are not identical, since some of the mount points include binaries that are also platform dependent.

Fig.8 shows the memory use of FLEDGE using either Docker or Containerd. Again, the ARM versions are much more resource efficient, using up to 50% less memory for Docker and 65% for Containerd.

As far as container runtimes go, Containerd is by far the best option to use with FLEDGE. The ARM setup of FLEDGE using Containerd requires only about 80MiB storage and 50MiB memory in total, including a VPN client container.

6.2 Virtual Kubelet integration

The effects of integrating the Virtual Kubelet into FLEDGE are shown in Fig.9 and Fig.10. In Section 3, Eq. 5 and Eq. 6 were constructed to calculate the maximum amount of storage and memory this integrated solution should use. By measuring the resource consumption of Virtual Kubelet pods on the master node, M_{Pod} is determined to be 10MiB and M_{Shr} 20MiB. Other factors are harder to pin down, but they are estimated at 500MiB for M_{Kube}, 0MiB for S_{Pod}, 40MiB for S_{Shr} and 1200MiB for S_{Kube}. Using the default Kubernetes node and pod limits, M_E and S_E are calculated and shown in the figures as horizontal lines, indicating the useful limits for memory and storage respectively.

As Fig.9 shows, integrating the Virtual Kubelet into FLEDGE is not optimal for storage, especially in the case of x64, but considering that it only goes 3MiB over the "limit" it is unlikely to matter much. Fig.10 shows slightly better results for memory use. On ARM, there is a good reason to run the Virtual Kubelet in FLEDGE on the edge, since it uses about 10% less memory than the calculated useful limit. For x64, moving the Virtual Kubelet to the edge is more or less memory neutral, with median memory use being exactly the limit.

6.3 Orchestrator comparison

Fig.11 shows the storage requirements of FLEDGE compared to those of Kubernetes. For both x64 and ARM, FLEDGE requires significantly less storage than Kubernetes, but the difference is largest on x64 with about 75% less storage. On an ARM device, FLEDGE requires about 60%

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

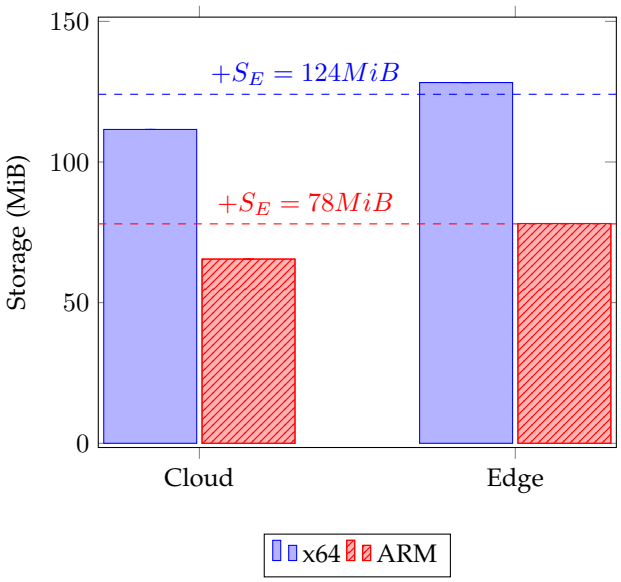


Fig. 9: Storage requirements of FLEDGE while running the Virtual Kubelet in the cloud or on the edge. The horizontal lines indicate the useful upper limits for integrating the Virtual Kubelet into FLEDGE on the edge for x64 and ARM, calculated by adding the result of Eq. 6 to the numbers of the *Cloud* category.

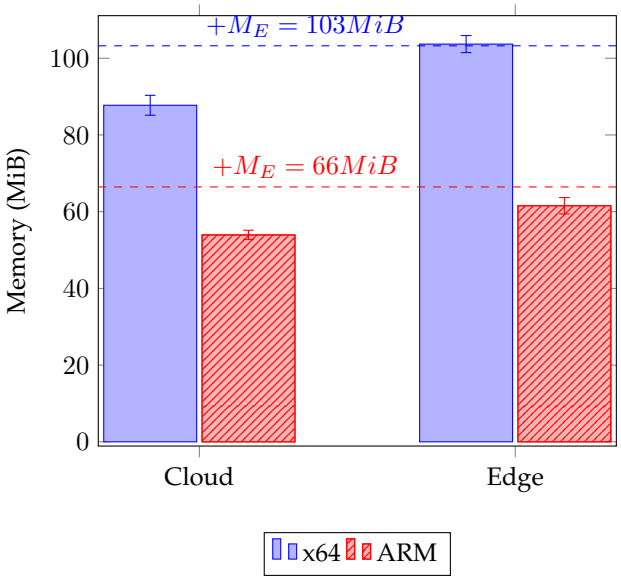


Fig. 10: Memory use of FLEDGE while running the Virtual Kubelet in the cloud or on the edge. The horizontal lines indicate the useful upper limits for integrating the Virtual Kubelet into FLEDGE on the edge for x64 and ARM, calculated by adding the result of Eq. 5 to the medians of the *Cloud* category.

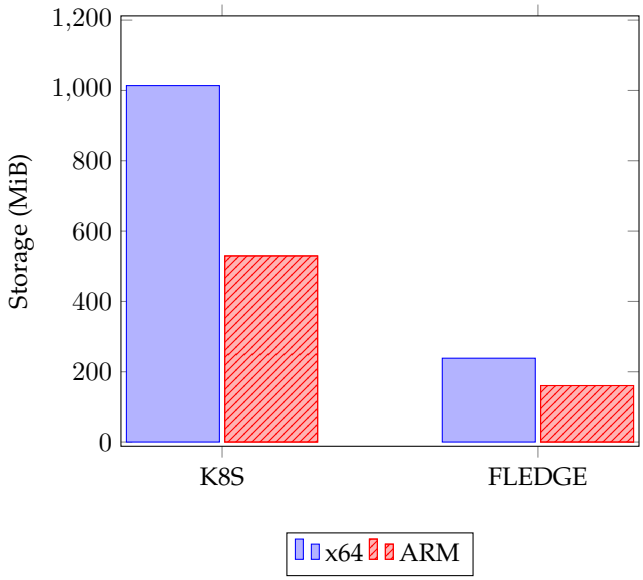


Fig. 11: Comparison of the storage requirements of Kubernetes and FLEDGE, both running a kube-proxy deployment.

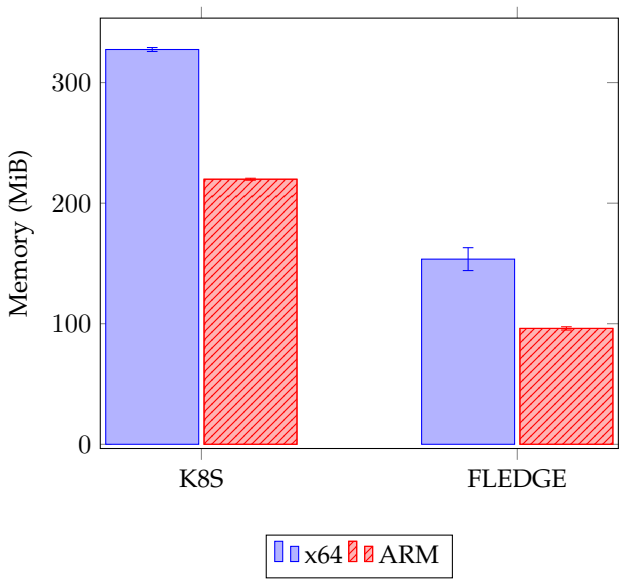


Fig. 12: Comparison of the memory use of Kubernetes and FLEDGE, both running a kube-proxy deployment.

less storage than Kubernetes. This large difference can be attributed to several factors, including the choice of Containerd over Docker and integrating several plugins instead of running them as containers.

The memory use of FLEDGE compared to Kubernetes is shown in Fig.12. Again, FLEDGE requires significantly fewer resources than Kubernetes, with both the x64 and ARM versions requiring around 50% less memory than Kubernetes. It is worth noting that simply eliminating Flannel in favor of a custom container networking solution saves around 24MiB of memory on ARM devices and 36MiB on x64.

These results show that FLEDGE, while remaining Kubernetes compatible, uses much less resources and is a viable container orchestrator for edge devices.

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

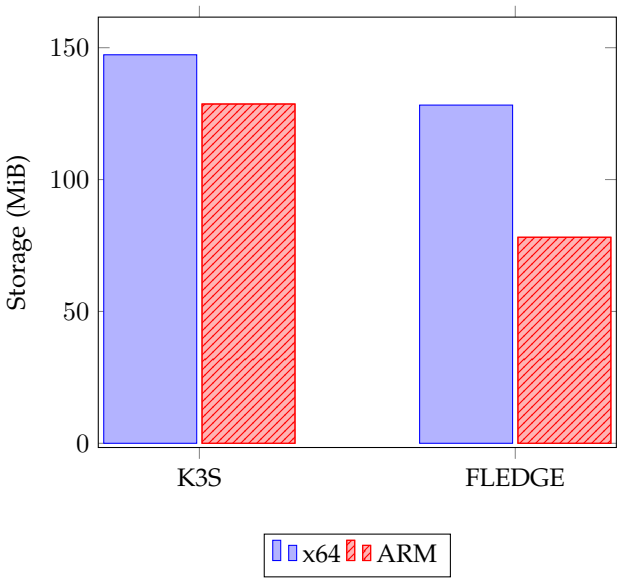


Fig. 13: Comparison of the storage requirements of K3S and FLEDGE, without kube-proxy.

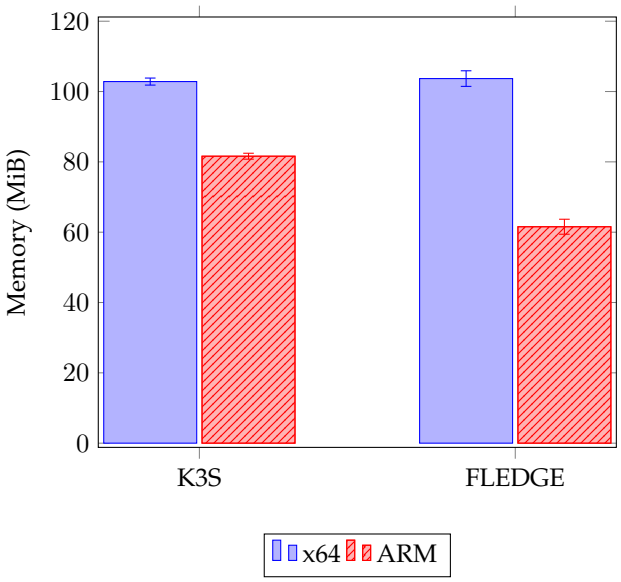


Fig. 14: Comparison of the memory use of K3S and FLEDGE, without kube-proxy.

The difference between K3S and FLEDGE, shown in Fig.13 for storage and Fig.14 for memory, is less impressive. However, FLEDGE still uses about 10% less storage than K3S on x64, and around 30% less on ARM. As far as memory goes, FLEDGE and K3S require more or less equal amounts on x64, but FLEDGE uses 25% less on ARM devices. Combined with the ability of FLEDGE to join Kubernetes clusters, which K3S cannot do, this makes a strong case for using FLEDGE as an edge container orchestrator compared to alternative software.

Finally, Fig.15 shows the amount of memory used for each container orchestrator, without any additional processes. Only in the case of Kubernetes has Flannel been included, since K3S and FLEDGE have pod and container networking by default. This chart shows that while both K3S and FLEDGE require only around 30% of the resources of Kubernetes, FLEDGE is more efficient on ARM devices, while K3S is more efficient on x64.

7 FUTURE WORK

This article presents a fully operational container orchestrator for edge devices, but there are aspects of FLEDGE that can be improved.

First of all, the integration of the Virtual Kubelet on the edge is not ideal. While it is better than managing each FLEDGE agent with separate pods in the cloud, the ideal solution may be to create a single service in the cloud that can manage hundreds or thousands of FLEDGE agents, scaling up only as required. This approach would be optimal for resource requirements, but it would likely require a lot of processing power and create a single point of failure.

Only Docker and Containerd were considered as container runtimes for FLEDGE, but many others exist, including rkt [54] and CRI-O [55]. Docker and Containerd were chosen because they are widely supported and popular, but it is unknown if another container runtime could give better results.

As orchestrator compatibility goes, K3S and FLEDGE already use both the Kubernetes and Containerd APIs, so with a little extra work it may be possible to have FLEDGE connect to both Kubernetes and K3S clusters, even simultaneously.

While FLEDGE is built to be Kubernetes compatible, it is unknown if optional features such as distributed storage work properly at this point. For the envisioned use of FLEDGE on edge devices, this is not important, but it could prove a valuable addition in the future.

OpenVPN is used to build a homogeneous network environment for FLEDGE to operate in, but other VPN software exists that may be more stable or provide faster connection speeds. Possible alternatives include Tinc, WireGuard and ZeroTier.

In Eq. 5, C_M represents the relative cost of edge memory versus cloud memory. In this article, it is naively assumed to be 1, but studies on the relative cost of edge resources and cloud resources could be interesting for the further development of software and container placement strategies.

The version of Kubernetes used in this article is limited to a maximum of 5.000 nodes and 150.000 pods in total. While this is sufficient for cloud clusters, the maximum number of nodes in particular will be too low for edge clusters. These numbers are not hard-coded, but based on the performance of several subsystems, such as node synchronization and pod status updates. It may be possible to increase the maximum number of nodes by optimizing the configuration of Kubernetes and severely limiting the maximum number of pods on an edge node. Another solution is to federate a number of Kubernetes clusters using KubeFed [56], thereby reducing the impact of the limits of a single cluster.

8 CONCLUSION

In the introduction, a number of requirements are proposed for FLEDGE:

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

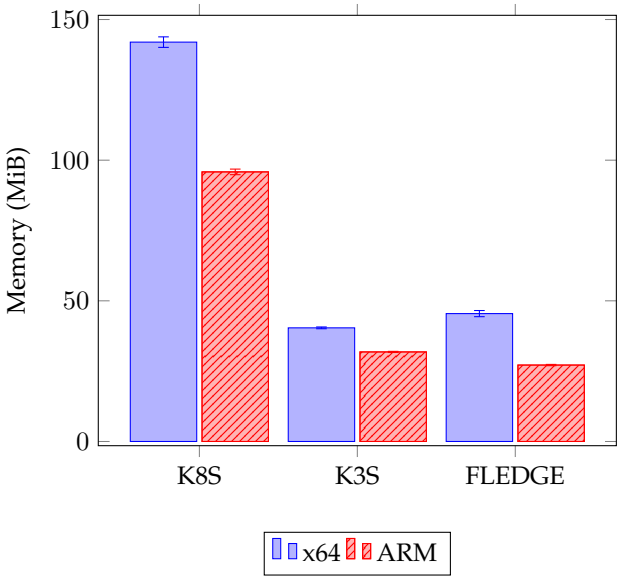


Fig. 15: Direct comparison of the memory use of the main process(es) of each container orchestrator. For Kubernetes, Flannel was included in the measurement because other orchestrators provide the same functionality by default.

- Secure communications between edge devices and the cloud by default, minimal impact on local networks.
- Compatibility with modern standards for container orchestration, or provide an adequate alternative.
- Low resource requirements, primarily in terms of memory but also in terms of processing and storage.

In this article, FLEDGE is shown to be a solution to these requirements by using Virtual Kubelets and agents on edge devices. A VPN is used to homogenize edge networks and provide a basic layer of security for communication with the cloud. Compatibility with container standards is achieved by using OCI [7] APIs to build FLEDGE. Some standards, such as CNI, can be safely ignored by using a custom implementation which does not impact the rest of the cluster. Low resource requirements are achieved partially by choosing Containerd as a container runtime, and partially through custom implementations of specific functionality such as CNI.

To further illustrate the low resource requirements of FLEDGE, several evaluations are performed. A FLEDGE setup is deployed using both Containerd and Docker, showing that FLEDGE using Containerd requires about half of the memory and storage of FLEDGE using Docker.

Similarly, the choice of running the Virtual Kubelets in the cloud or integrating them into FLEDGE agents is backed up by a theoretical model and an evaluation. The results show that it is preferable to integrate the Virtual Kubelets into FLEDGE agents, reducing overall complexity. On x64 platforms, the total amount of resources required is almost identical for the two solutions, but on ARM the results are slightly better when running everything on the edge.

Finally, Kubernetes and K3S are discussed as alternatives to FLEDGE, followed by an evaluation to compare them in terms of memory and storage requirements. The results show that FLEDGE requires only about 40-50% of the resources of a similar Kubernetes worker node, while it also requires 25% less resources than K3S on ARM devices. On x64 devices, FLEDGE and K3S resource requirements are nearly equal.

The conclusion is that FLEDGE, despite its experimental status, can deploy Kubernetes pods while using significantly less resources than other container orchestrators. Several topics for future work are discussed, mostly focused on improving FLEDGE itself.

REFERENCES

[1] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, mar 2015.

[2] M. Amaral, J. Polo, D. Carrera, I. Mohomed, M. Unuvar, and M. Steinder, "Performance evaluation of microservices architectures using containers," in *2015 IEEE 14th International Symposium on Network Computing and Applications*. IEEE, sep 2015.

[3] "What is kubernetes?" Apr. 2019. [Online]. Available: https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/

[4] M. Villari, M. Fazio, S. Dustdar, O. Rana, and R. Ranjan, "Osmotic computing: A new paradigm for edge/cloud integration," *IEEE Cloud Computing*, vol. 3, no. 6, pp. 76–83, nov 2016.

[5] "Why docker?" [Online]. Available: https://www.docker.com/why-docker

[6] H. Sahni, "Cnm vs cni," May 2017. [Online]. Available: https://www.nuagenetworks.net/blog/container-networking-standards/

[7] "About oci." [Online]. Available: https://www.opencontainers.org/about

[8] "Virtual kubelet," Apr. 2019. [Online]. Available: https://github.com/virtual-kubelet/virtual-kubelet

[9] P. Mach and Z. Becvar, "Mobile edge computing: A survey on architecture and computation offloading," *IEEE Communications Surveys & Tutorials*, vol. 19, no. 3, pp. 1628–1656, 2017.

[10] K. Kumar and Y.-H. Lu, "Cloud computing for mobile users: Can offloading computation save energy?" *Computer*, vol. 43, no. 4, pp. 51–56, apr 2010.

[11] J. Barrameda and N. Samaan, "A novel statistical cost model and an algorithm for efficient application offloading to clouds," *IEEE Transactions on Cloud Computing*, vol. 6, no. 3, pp. 598–611, jul 2018.

[12] S. Sarkar, S. Chatterjee, and S. Misra, "Assessment of the suitability of fog computing in the context of internet of things," *IEEE Transactions on Cloud Computing*, vol. 6, no. 1, pp. 46–59, jan 2018.

[13] D. Santoro, D. Zozin, D. Pizzolli, F. D. Pellegrini, and S. Cretti, "Foggy: A platform for workload orchestration in a fog computing environment," in *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*. IEEE, dec 2017.

[14] A. Morshed, P. P. Jayaraman, T. Sellis, D. Georgakopoulos, M. Villari, and R. Ranjan, "Deep osmosis: Holistic distributed deep learning in osmotic computing," *IEEE Cloud Computing*, vol. 4, no. 6, pp. 22–32, nov 2017.

[15] "Kubernetes federation," Mar. 2019. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/federation/

[16] T. Wauters, B. Vermeulen, W. Vandenberghe, P. Demeester, S. Taylor, L. Baron, M. Smirnov, Y. Al-Hazmi, A. Willner, M. Sawyer, D. Margery, T. Rakotoarivelo, F. Lobillo Vilela, D. Stavropoulos, C. Papagianni, F. Francois, C. Bermudo, A. Gavras, D. Davies, and S.-Y. Park, "Federation of internet experimentation facilities: architecture and implementation," in *IEEE European Conference on Networks and Communications 2014 (EuCNC'2014)*, 06 2014.

[17] R. Moreno-Vozmediano, E. Huedo, I. M. Llorente, R. S. Montero, P. Massonet, M. Villari, G. Merlino, A. Celesti, A. Levin, L. Schour, C. Vázquez, J. Melis, S. Spahr, and D. Whigham, "BEACON: A cloud network federation framework," in *Communications in Computer and Information Science*. Springer International Publishing, 2016, pp. 325–337.

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

[18] P. Bottoni, E. Gabrielli, G. Gualandi, L. V. Mancini, and F. Stolfi, "FedUp! cloud federation as a service," in *Service-Oriented and Cloud Computing*. Springer International Publishing, 2016, pp. 168–182.
[19] T. Goethals, D. Kerkhove, L. V. Hoye, M. Sebrechts, F. D. Turck, and B. Volckaert, "FUSE: A microservice approach to cross-domain federation using docker containers," in *Proceedings of the 9th International Conference on Cloud Computing and Services Science*. SCITEPRESS - Science and Technology Publications, 2019.
[20] D. Puthal, S. Nepal, R. Ranjan, and J. Chen, "Threats to networking cloud and edge datacenters in the internet of things," *IEEE Cloud Computing*, vol. 3, no. 3, pp. 64–71, may 2016.
[21] M. Villari, M. Fazio, S. Dustdar, O. Rana, L. Chen, and R. Ranjan, "Software defined membrane: Policy-driven edge and internet of things security," *IEEE Cloud Computing*, vol. 4, no. 4, pp. 92–99, jul 2017.
[22] N. Chowdhury and R. Boutaba, "Network virtualization: State of the art and research challenges," *IEEE Communications Magazine*, vol. 47, no. 7, pp. 20–26, jul 2009.
[23] H. Hamed, E. Al-Shaer, and W. Marrero, "Modeling and verification of IPSec and VPN security policies," in *13TH IEEE International Conference on Network Protocols (ICNP'05)*. IEEE, 2005.
[24] F. Pohl and H. D. Schotten, "Secure and scalable remote access tunnels for the IIoT: An assessment of openVPN and IPsec performance," in *Service-Oriented and Cloud Computing*. Springer International Publishing, 2017, pp. 83–90.
[25] I. Kotuliak, P. Rybar, and P. Truchly, "Performance comparison of IPsec and TLS based VPN technologies," in *2011 9th International Conference on Emerging eLearning Technologies and Applications (ICETA)*. IEEE, oct 2011.
[26] A. Buzachis, A. Galletta, L. Carnevale, A. Celesti, M. Fazio, and M. Villari, "Towards osmotic computing: Analyzing overlay network solutions to optimize the deployment of container-based microservices in fog, edge and IoT environments," in *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*. IEEE, may 2018.
[27] "Cluster networking," Apr. 2019. [Online]. Available: https://kubernetes.io/docs/concepts/cluster-administration/networking/
[28] R. Moreno-Vozmediano, R. S. Montero, E. Huedo, and I. M. Llorente, "Cross-site virtual network in cloud and fog computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 46–53, mar 2017.
[29] L. Xu, J. Lee, S. H. Kim, Q. Zheng, S. Xu, T. Suh, W. W. Ro, and W. Shi, "Architectural protection of application privacy against software and physical attacks in untrusted cloud environment," *IEEE Transactions on Cloud Computing*, vol. 6, no. 2, pp. 478–491, apr 2018.
[30] C. Pahl and B. Lee, "Containers and clusters for edge cloud architectures – a technology review," in *2015 3rd International Conference on Future Internet of Things and Cloud*. IEEE, aug 2015.
[31] C. Dupont, R. Giaffreda, and L. Capra, "Edge computing in IoT context: Horizontal and vertical linux container migration," in *2017 Global Internet of Things Summit (GIoTS)*. IEEE, jun 2017.
[32] "Rancher labs - k3s lightweight kubernetes." [Online]. Available: https://k3s.io/
[33] "Microk8s." [Online]. Available: https://microk8s.io/
[34] Y. Xiong, Y. Sun, L. Xing, and Y. Huang, "Extend cloud to edge with KubeEdge," in *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, oct 2018.
[35] S. Hoque, M. S. de Brito, A. Willner, O. Keil, and T. Magedanz, "Towards container orchestration in fog computing infrastructures," in *2017 IEEE 41st Annual Computer Software and Applications Conference (COMPSAC)*. IEEE, jul 2017.
[36] "Kubernetes - custom resources." [Online]. Available: https://kubernetes.io/docs/concepts/extend-kubernetes/api-extension/custom-resources/
[37] "Deploy iot edge gateway on kubernetes." [Online]. Available: https://docs.microsoft.com/en-us/samples/azure-samples/iotedge-gateway-on-kubernetes/iot-edge-workloads-on-kubernetes/
[38] "Google cadvisor," Aug. 2018. [Online]. Available: https://github.com/google/cadvisor
[39] "Heapster is now retired," Nov. 2018. [Online]. Available: https://github.com/kubernetes-retired/heapster
[40] "Kubernetes resource metrics api," May 2018. [Online]. Available: https://github.com/kubernetes/metrics
[41] M. GroBmann and C. Klug, "Monitoring container services at the network edge," in *2017 29th International Teletraffic Congress (ITC 29)*. IEEE, sep 2017.
[42] S. Tarkoma, *Overlay Networks*. Auerbach Publications, feb 2010.
[43] "Kubernetes - building large clusters." [Online]. Available: https://kubernetes.io/docs/setup/cluster-large/
[44] A. Verma, L. Pedrosa, M. Korupolu, D. Oppenheimer, E. Tune, and J. Wilkes, "Large-scale cluster management at google with borg," in *Proceedings of the Tenth European Conference on Computer Systems - EuroSys '15*. ACM Press, 2015.
[45] "k3s - 5 less than k8s," May 2019. [Online]. Available: https://github.com/rancher/k3s
[46] "Kubeedge: A kubernetes native edge computing framework," 2019. [Online]. Available: https://kubeedge.io/en/
[47] "What is kubeedge: Architecture," 2019. [Online]. Available: https://docs.kubeedge.io/en/latest/modules/kubeedge.html #architecture
[48] R. A. Light, "Mosquitto: server and client implementation of the MQTT protocol," *The Journal of Open Source Software*, vol. 2, no. 13, p. 265, may 2017.
[49] "The df command." [Online]. Available: https://www.linuxjournal.com/article/2747
[50] "pmap - report memory map of a process." [Online]. Available: https://linux.die.net/man/1/pmap
[51] "Propertional set size (pss)." [Online]. Available: http://lkml.iu.edu/hypermail/linux/kernel/0708.1/3930.html
[52] "Docker components explained." [Online]. Available: http://alexander.holbreich.org/docker-components-explained/
[53] "kube-proxy." [Online]. Available: https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/
[54] "Getting started with rkt." [Online]. Available: https://coreos.com/rkt/docs/latest/getting-started-guide.html
[55] "Cri-o, lightweight container runtime for kubernetes." [Online]. Available: https://cri-o.io/
[56] "Kubefed - kubernetes cluster federation." [Online]. Available: https://github.com/kubernetes-sigs/kubefed

Tom Goethals received the masters degree in Information Engineering Technology from University College Ghent, Belgium in 2013. After several years as a software engineer, he joined the Internet Technology and Data Science Laboratory (IDLab) at Ghent University-imec in 2018 to pursue a Ph.D. His current research deals with scalable and reliable software systems for Smart Cities, working on various projects in cooperation with industry partners.

Filip De Turck leads the network and service management research group at the Department of Information Technology of the Ghent University, Belgium and imec. He (co-) authored over 500 peer reviewed papers and his research interests include telecommunication network and service management, and design of efficient virtualized network and cloud systems. In this research area, he is involved in several research projects with industry and academia, serves as Chair of the IEEE Technical Committee on Network Operations and Management (CNOM), and is on the TPC of many network and service management conferences and workshops. Prof. Filip De Turck serves as Editor in Chief of IEEE Transactions of Network and Service Management (TNSM), and steering committee member of the IEEE Conference on Network Softwarization (IEEE NetSoft).

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.

This is the author's version of an article that has been published in this journal. Changes were made to this version by the publisher prior to publication.
The final version of record is available at http://dx.doi.org/10.1109/TCC.2020.3033807

Bruno Volckaert is professor advanced programming and software engineering in the Department of Information Technology (INTEC) at Ghent University and senior researcher at imec. He obtained his Master of Computer Science degree in 2001 from Ghent University, after which he worked on his PhD at Ghent University on data intensive scheduling and service management for Grid computing, which he obtained in 2006. His current research deals with reliable and high performance distributed software systems for City-of-Things (IoT for Smart Cities), distributed decision support systems, intelligent transportation applications and autonomous optimization of cloud-based applications. He has worked on over 45 national and international research projects and is author or co-author of more than 120 papers published in international journals and conference proceedings.

Copyright (c) 2020 IEEE. Personal use is permitted. For any other purposes, permission must be obtained from the IEEE by emailing pubs-permissions@ieee.org.