FLEDGE: Kubernetes Compatible Container Orchestration on Low-Resource Edge Devices

Tom Goethals(✉), Filip De Turck, and Bruno Volckaert

imec, IDLab, Department of Information Technology, Ghent University, Technologiepark-Zwijnaarde 15, 9052 Gent, Belgium
togoetha.goethals@UGent.be

Abstract. In recent years, containers have quickly gained popularity in the cloud, mostly thanks to their scalable, ethereal and isolated nature. Simultaneously, edge devices have become powerful enough to run containerized microservices, while remaining small and low-powered. These evolutions have triggered a wave of research into container placement strategies on clusters including edge devices, leading to concepts such as fog computing. These container placement strategies can optimize workload placement across cloud and edge clusters, but current container orchestrators are very resource intensive and are not designed to run on edge devices.

This paper presents FLEDGE as a Kubernetes compatible edge container orchestrator. A number of aspects of how to achieve low-resource container orchestration are examined, for example the choice of container runtime and how to implement container networking. Finally, a number of evaluations are performed, comparing FLEDGE to K3S and Kubernetes, to show that it is a viable alternative to existing container orchestrators.

Keywords: Edge networks · Edge computing · Container orchestration · Containers · VPN

1 Introduction

In recent years, containers have quickly gained popularity for cloud applications, thanks to their limited resource requirements and fast spin-up times compared to virtual machines [1]. The complexity of managing large amounts of containers has led to container orchestrators such as Kubernetes [2], which handles the deployment and scaling of containerized services.

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 lead to research aimed at deploying containers on edge devices, and shifting containerized workloads between the cloud and the edge. Most container orchestrators are designed to run in the cloud, and are very flexible and modular but not very mindful of resource consumption. Edge devices on the other hand, are typically low-resource devices and non-extensible, especially in terms of memory.

Additionally, container deployments in the cloud are often generic, scalable microservices, whereas those on the edge will be more suited to local computing, with less focus on scaling. This means that an edge container orchestrator should be primarily built to use minimal resources, and less for constantly moving and migrating containers.

Edge devices are often located in networks with potentially less focus on security and organization. In many cases, the device is hidden behind a router with a firewall or NAT, and IP addresses and port mappings are unpredictable.

Being designed for the cloud, most container orchestrators expect a well-organized and homogeneous infrastructure, where all network resources are predictable and controlled. Additionally, unlike intra-cloud communication, communication outside the cloud could be intercepted very easily, so all communication between the cloud and containers deployed on the edge should be secured by default. Any solution to deploy containers on edge devices should therefore not only create a heterogeneous and predictable networking environment for containers to operate in, but also secure communication with the cloud by default.

Continued development of container management tools such as Kubernetes and Docker [3] has led to the development of a number of standards, for example the Container Network Interface (CNI [4]) for container networking, and container format standards from the Open Container Initiative (OCI [5]).

Any solution for edge container deployment should be compatible with existing container standards, so it is important that they are implemented to the extent possible on edge devices. If any standards are ignored or not fully implemented, it should not affect the rest of the cluster.

The requirements for a good container orchestrator for edge devices can therefore be summarized as:

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

This paper presents FLEDGE as a low-resource container orchestrator which is capable of directly connecting to Kubernetes clusters using modified Virtual Kubelets [6] and a VPN.

Section 2 presents existing research related to the topics in this introduction. Section 3 shows how FLEDGE meets the requirements put forward in this introduction, while Sect. 4 discusses alternative edge container orchestrators. In Sect. 5, an evaluation setup and methodology are presented to compare FLEDGE to alternative orchestrators. The results of the evaluations are presented and discussed in Sect. 6, with suggestions for future work in Sect. 7. Finally, Sect. 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 edge offloading [7], cloud offloading [8], and osmotic computing [9]. Many studies exist on different container placement strategies, from simple but effective resource requests and grants [10], to using deep learning for allocation and real-time adjustments [11].

Kubernetes is capable of forming federations of Kubernetes clusters [12], but this paper 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 [13], Beacon [14], FedUp! [15] and FUSE [16]. 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 [17] which identifies possible threats, and [18] which proposes a Software Defined Membrane as a novel security paradigm for all aspects of microservices.

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 [19]. Some studies deal with the security aspects of a VPN [20], while many others focus on the throughput performance of VPNs [21,22].

A study by Pahl et al. [23] gives a general overview of how to create edge cloud clusters using containers. While FUSE [16] is capable of deploying Kubernetes worker nodes on edge devices, the resulting framework is too resource-intensive for most edge hardware. Cloud4IoT [24] 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 [25], 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 cannot directly connect to Kubernetes clusters. KubeEdge [26] is a recent development, aiming to extend Kubernetes to edge clusters. Despite being based on Kubernetes, it is not directly compatible with Kubernetes master nodes and needs an extra cloud component to function properly.

3 FLEDGE

This section gives an overview of what a Virtual Kubelet is and how FLEDGE, written in Golang, builds on it to meet the requirements stated in the introduction.

A Virtual Kubelet is a small service which acts as a proxy for Kubernetes to any platform that can run containers, for example Amazon AWS, Microsoft Azure or edge devices. It registers itself as a node in Kubernetes and passes API calls from Kubernetes to brokers which translate those calls to the container platform they implement. These API calls include pod management, pod status, node status, logging and metrics.

The concepts of FLEDGE are shown in Fig. 1, with the Virtual Kubelet acting as a proxy to the FLEDGE broker. The FLEDGE broker is responsible for sending API calls to the edge, where they are decomposed into container networking, cgroup and namespace management, and container deployments. This collection of components will be referred to as a FLEDGE agent.

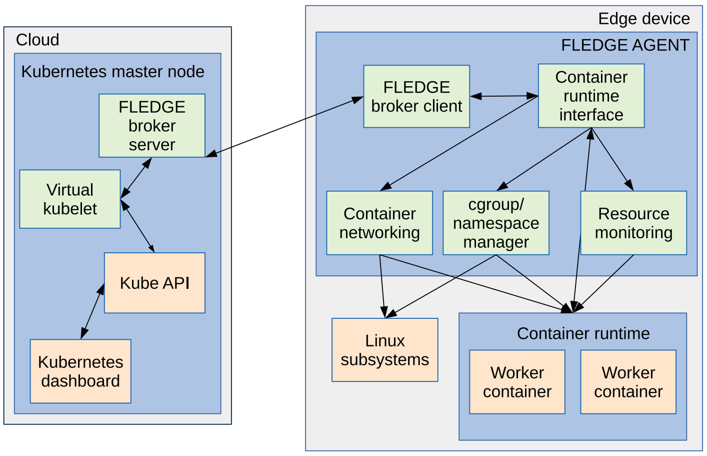


Fig. 1. Conceptual overview of FLEDGE and its use of a Virtual Kubelet.

3.1 Compatibility

One of the stated requirements for FLEDGE is compatibility with existing container standards and runtimes.

Both Docker and Containerd are popular container runtimes and both support OCI standards, and by extension Docker containers. Furthermore, since version 1.11, Docker uses Containerd as an underlying container runtime. In terms of compatibility, both runtimes are good choices, so the decision will be up to resource requirements, as discussed in Sect. 3.3.

Another aspect of compatibility is container networking. In Kubernetes, the master node makes high-level decisions on container networking, such as which IP range to assign to individual nodes. These decisions are relayed to CNI compatible plugins (eg. Flannel, Weave) on worker nodes, which translate the high-level decisions into low-level network configuration.

FLEDGE does this differently by fulfilling the role of both Kubelet and container networking plugin. The number of pods deployed on an edge node is likely to be low due to resource constraints, so the container networking handler (Fig. 1 Container networking) can be simple and naive, assigning pods the first free IP address in its range. The same handler also makes sure network namespaces are configured correctly.

By default, Kubernetes will not attempt to deploy Flannel on a FLEDGE agent. However, because the FLEDGE agent uses the IP range assigned to it by Kubernetes, the rest of the cluster will still be able to reach pods deployed on it. This means that this approach is sufficient, despite not implementing CNI.

3.2 Security and Stability

Edge devices, especially consumer grade, often operate in networks with little to no security and organization. Not only may the devices find themselves in unexpected topologies with random IP address assignments and unknown port mappings, they may also be stuck behind a router with NAT or a firewall. Additionally, while traffic between Kubernetes and its Kubelets is secured by default, this is not always true for services deployed on worker nodes.

In FLEDGE, these issues are solved by connecting edge nodes to the cloud using OpenVPN and building a container network on top of it, as shown in Fig. 2. Using a VPN ensures that all ports are available and open and IP address assignments are logical and reachable by the cloud. Furthermore, the physical network of the device no longer matters, the virtual network can be organized according to any parameters. Finally, traffic between the edge and the cloud is encrypted by default, providing a basic layer of security.

However, there are some downsides to this approach. Using OpenVPN, especially with encryption, is a drain on system and network resources, likely reducing the scalability of the cluster. Moreover, VPN overhead may have a significant impact on edge devices, which have limited computational power. Anyone with physical access to the device can still gain access to the system, and possibly even cloud resources through the VPN. This problem is exacerbated because like Kubernetes and K3S, the FLEDGE agent requires root access to run properly, so hardware security and OS-level security are required to prevent these problems.

Figure 3 gives an overview of the different networks involved in a setup with FLEDGE nodes. Green arrows indicate traffic flows allowed by FLEDGE, while red ones indicate traffic flows forbidden by default. This shows that all devices in the VPN or the Kubernetes pod network can reach each other, but other devices can only be reached by being in the same physical network.

Container images may contain software or data that needs to be protected from unauthorized access. Both the FLEDGE agent and container runtimes

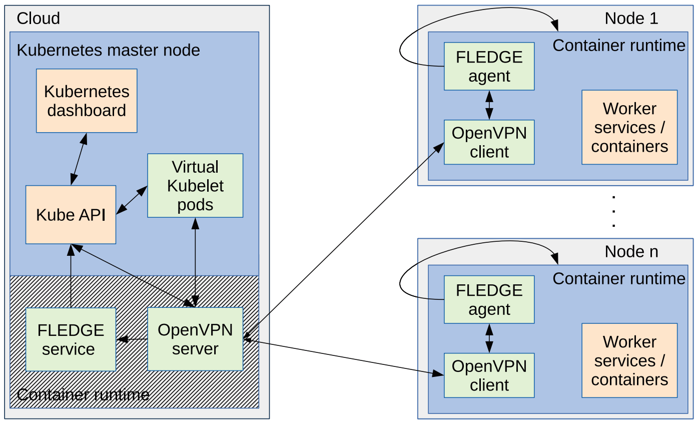


Fig. 2. High-level overview of network traffic flow of FLEDGE, using OpenVPN to connect edge nodes to the cloud.

could potentially be abused to access container images, but some steps can be taken to mitigate this.

Containers and pods are assigned different file system namespaces by container runtimes. While the root user can still access these namespaces, root login can be disabled and the file systems can be protected from other users. To minimize chances of container images being copied, and to avoid clutter, they can be removed when no longer running. However, this will increase the time required for redeployment of containers, thereby affecting performance. Finally, FLEDGE cleans up all containers, images and network infrastructure on shutdown.

3.3 Low Resource Use

An important choice for low resource use is the container runtime. As Sect. 3.1 showed, both Docker and Containerd are good choices in terms of compatibility. However, as Docker actually relies on Containerd since version 1.11, Containerd is likely the more resource-friendly option. This choice will be further evaluated in Sect. 6.

The choice for a custom container networking solution in FLEDGE is optimal in terms of resource requirements. While normal CNI plugins for Kubernetes are run as containers, a flexible and durable approach, they also require more resources than simply embedding container networking into the orchestrator process.

Both namespace and cgroup handling have been implemented in FLEDGE. While FLEDGE relies on the container runtime to set up the namespaces and cgroups setup for the first container of a pod, it reuses those namespaces for all other containers in the same pod. This approach is compatible with both Docker and Containerd, and has the added benefits of being very simple.

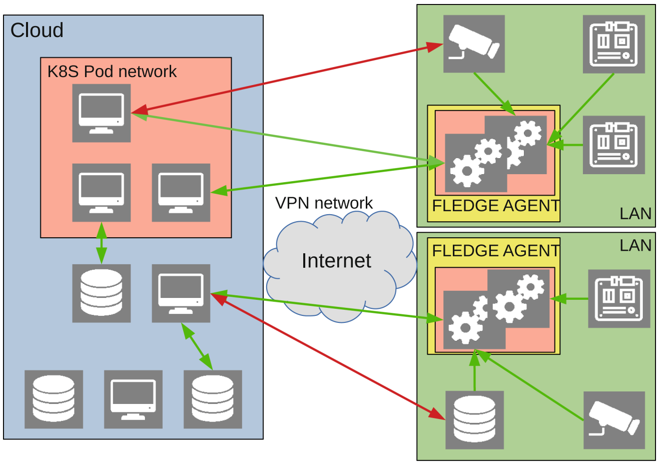


Fig. 3. Overview of network traffic flows in a cluster using FLEDGE nodes. Green arrows indicate possible traffic flows. (Color figure online)

4 Alternatives

This section discusses some alternative container orchestrators, giving a short history and possible advantages and disadvantages for each, which will then be compared to FLEDGE in terms of resource requirements.

4.1 Kubernetes

Kubernetes [2] is a widely used container orchestrator originally inspired by Google Borg [27]. Due to its popularity and extensive development, it has contributed to several container standards. Because it is made to run in the cloud, it is very flexible. However, as Sect. 6 shows it also uses a lot of resources, making it hard to use on edge devices.

An important difference between FLEDGE and Kubernetes is that the latter requires all swapping to be disabled in order to run, which can cause serious problems on devices with limited memory. FLEDGE has no such requirement, allowing all memory subsystems to perform as intended.

4.2 K3S

K3S [25] is a novel container orchestrator based on Kubernetes, modified specifically for edge devices. Version 0.1.0 was released in February 2019, while the version used for the evaluation is v0.3.0 from March 2019. K3S has its own master nodes, unlike FLEDGE which connects to Kubernetes master nodes.

Unlike FLEDGE, which starts from scratch and builds around Kubernetes compatibility, K3S starts with the full Kubernetes source code and eliminates deprecated or little-used functionality. Like FLEDGE, it prefers to hard wire certain types of functionality. For example, it uses Flannel for container networking and forces the use of Containerd.

While being built from the full Kubernetes source means K3S has excellent support for standards, this may also be a disadvantage in terms of resource requirements. It also has its own join mechanism and is, for the moment, incompatible with Kubernetes master nodes, so it cannot directly connect to existing Kubernetes clusters.

4.3 KubeEdge

KubeEdge [28] is an early stage Edge Computing Framework built on Kubernetes and Docker. Its first release was in December 2018, with version 0.3.0 being released as of May 2019. It consists of a cloud part and an edge part [29], with the cloud part interfacing with the cloud Kubernetes API and taking care of node management. The edge part is deployed on each individual device and takes care of pod and low-level facility management.

While its functions include deploying Kubernetes pods on edge networks, it aims to be an entire ecosystem for edge computing, including storage and event-based communication based on MQTT [30]. Because it is hard to isolate the container orchestration part, KubeEdge will not be evaluated in this paper. However, since it uses Docker, it is unlikely to be resource efficient, a point which will be proven in Sect. 6.

5 Evaluation Setup

With the most important concepts of FLEDGE explained and alternative orchestrators discussed, an evaluation environment can be set up and a number of evaluations can be performed. These are intended to validate the choice of container runtime and compare FLEDGE to K3S v0.3.0 and Kubernetes v1.14 in terms of resource requirements. The source code of FLEDGE is made available on Github[1].

5.1 Methodology

Figure 4 shows the hardware setup used for the evaluations, which is run on the imec Virtual Wall [31].

The VWall master node fulfills the role of K3S/Kubernetes master node. Because FLEDGE is aimed at worker nodes, the specifications and performance of this node are not important.

The VWall server (x64) is used to determine the resource requirements of each orchestrator on an x64 worker node. This device runs Ubuntu 18.04 and has an AMD Opteron 2212 processor at 2 GHz and 4 GiB RAM.

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

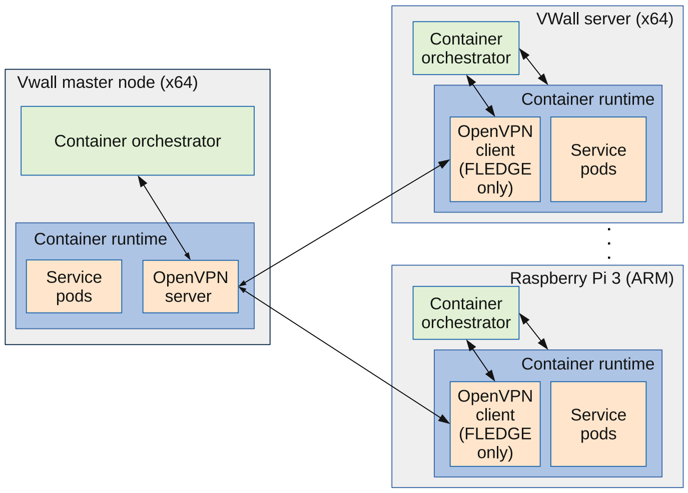


Fig. 4. Overview of the hardware setup used for the evaluation. Note that the OpenVPN containers are only used by FLEDGE, other orchestrators connect directly to the master node via LAN.

The Raspberry Pi 3 is used to evaluate each orchestrator on an ARM device, specifically armhf. This device runs Raspbian with kernel version 4.14.98-v7+ on the default hardware configuration, specifically 1 GiB RAM and a quad-core 1.2 GHz CPU. All devices are in the same geographical location and are connected by Gigabit LAN (100 Mbps max for Raspberry Pi 3). The OpenVPN server and clients are only used when FLEDGE is deployed on the worker nodes. Kubernetes and K3S connect to the master node directly via LAN. All evaluations will be run on both armhf and x64.

For Kubernetes, Docker is used, while Containerd is required by K3S. The container runtime used by FLEDGE is specified in each evaluation.

Storage requirements are measured using the df command [32], both before and after orchestrator setup. This approach takes not only the orchestrator into account, but all dependencies and libraries as well. To ensure proper measurements, the devices are wiped after each run.

Determining memory use is more complex than measuring storage requirements. Unlike the myriad files involved in a container orchestrator, the process running it are more easily identified, allowing for precise and detailed measurements. During orchestrator setup some processes will require memory, used to launch containers or initialize facilities, which is later released. This means that memory use must be monitored for a significant amount of time.

Processes can have private and shared memory. Measuring both memory sets is easy, but a fair method is needed to calculate the exact amount of memory used by each process.

Taking the above into account, each evaluation measures the memory use of a set of processes every 30 s, over a period of 15 min. The pmap [33] command is used to determine the Proportional Set Size (PSS) [34] of each process, which is calculated according to:

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

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

This evaluation aims to show that the choice of container runtime can have a large impact on the resource requirements of a container orchestrator. To verify this and determine the best choice, FLEDGE is set up using both Docker and Containerd.

To avoid interference from other containers, no pods or containers are deployed other than the FLEDGE agent and a VPN container. To determine the overhead of containerizing FLEDGE, a third case is evaluated in which the FLEDGE agent runs as a host service instead of being deployed as a container.

The processes monitored for this evaluation are the container runtime, the FLEDGE agent, the VPN client container and container shims [35].

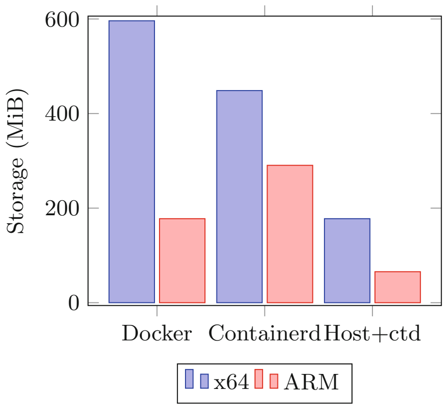
5.3 Orchestrator Comparison

In order to verify that FLEDGE is a low-resource solution for edge container orchestration, this evaluation compares it against Kubernetes and K3S. In the Kubernetes comparison, Flannel is used as a CNI plugin and the master node is allowed to deploy kube-proxy [36] on the edge node. Since FLEDGE has its own container networking, Flannel will not be deployed on the FLEDGE edge node. In the K3S comparison, no kube-proxy will be deployed on FLEDGE.

In this evaluation, FLEDGE is run as a host service and uses Containerd as a container runtime. The monitored processes are the container orchestrator, container runtime, shims and any deployed containers.

6 Results

This section presents the results of the evaluations described in Sect. 5. While the results for storage requirements are simple bar charts representing the median case, the results for memory consumption are more dynamic, including whiskers for the median absolute deviation.



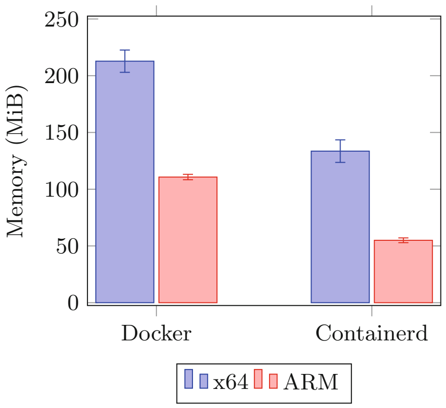


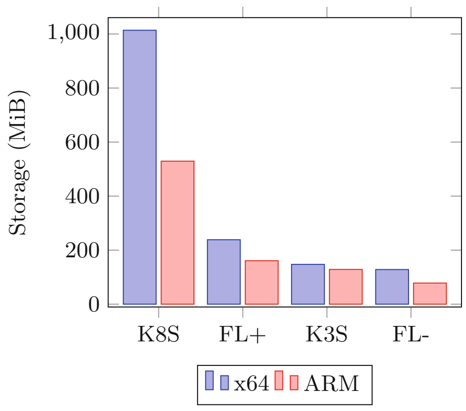
Fig. 5. Storage and memory requirements of FLEDGE using different container runtimes. The Host+ctd category shows the results for FLEDGE running as a host service.

6.1 Container Runtime

Figure 5 shows the storage requirements for FLEDGE deployments with Docker and Containerd.

An important observation is that in all cases, FLEDGE requires significantly less storage on ARM than it does on x64, though the exact amount varies. The combination of FLEDGE and Docker, for example, requires 3 times as much storage on x64 as it does on ARM. While the results suggest that Containerd is much less efficient on ARM than Docker, these numbers conflict with the fact that Docker uses Containerd to run containers. The reason for this is rooted in how Containerd and Docker handle container filesystems and mounts. In order for a containerized FLEDGE agent to be able to deploy containers on Containerd itself, many directories and files need to be mounted into the FLEDGE agent container. It turns out that Containerd mounts have a lot of overhead, resulting in the large container filesystem shown in Fig. 5. To validate this theory, another evaluation was done by deploying a FLEDGE agent as a host service, shown as "Host+ctd". The Containerd installation for this evaluation was also optimized, resulting in a 73 MiB size reduction on x64, and a 14 MiB reduction on ARM. The result is very resource efficient, at the cost of not having the FLEDGE agent isolated in a container. Note that this same approach does not affect Docker much, indicating that while it may use Containerd as a runtime, it has a more efficient method of handling file system layers.

Figure 5 also shows the memory consumption of FLEDGE deployments with Docker and Containerd. The ARM versions are again much more efficient, using up to 50% less memory than x64 in the case of Docker and 65% in the case of Containerd. The results show that Containerd is by far the best container runtime to use with FLEDGE. When running FLEDGE as a host service, the



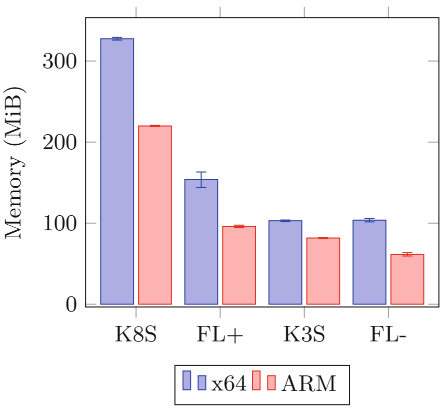


Fig. 6. Storage and memory requirements of evaluated container orchestrators. FL+ and FL- indicate FLEDGE running with and without kube-proxy, respectively.

total resource requirements are only 80 MiB storage and 50 MiB memory on ARM, including a VPN client container.

6.2 Orchestrator Comparison

Figure 6 shows the storage requirements for all container orchestrators. Note that FLEDGE is included twice in this chart; with a kube-proxy (“FL+”) and without a kube-proxy (“FL-”). Considering functionality, it is best to compare Kubernetes to FLEDGE with a kube-proxy, and K3S to FLEDGE without a kube-proxy.

Compared to Kubernetes, FLEDGE (“FL+”) only needs around 25% as much storage on x64 and 40% on ARM. This large difference can be attributed to many factors, including the choice of Containerd over Docker and the integration of several plugins instead of running them as containers.

When comparing FLEDGE (“FL-”) to K3S, the difference is smaller than with Kubernetes, but still significant. FLEDGE requires about 10% less storage on x64, and around 30% less on ARM.

The results for memory consumption are shown in Fig. 6, using the same notation for FLEDGE with and without kube-proxy. These results are less spread out than those of the storage requirements.

For starters, FLEDGE only requires about half as much memory as Kubernetes on both x64 and ARM. Note that simply eliminating Flannel and implementing a custom container networking solution already saves 36 MiB of memory on x64 and 24 MiB on ARM, or around 10% of Kubernetes’ memory consumption.

Compared to K3S, FLEDGE has similar memory consumption on x64, but around 25% less on ARM. Considering that most IoT/edge devices are ARM based, this is a significant improvement.

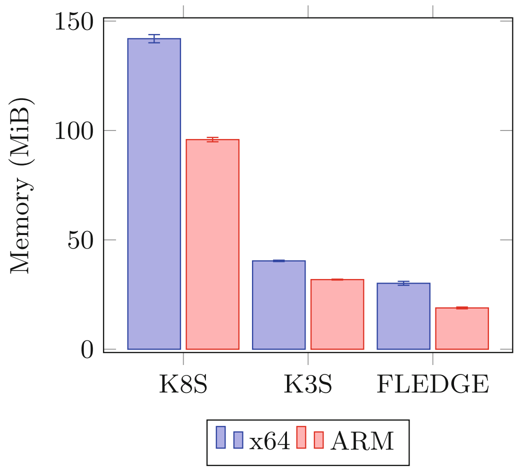


Fig. 7. Memory consumption of the main process of each container orchestrator. For Kubernetes, Flannel was included in the measurement because other orchestrators provide a container network by default.

Finally, Fig. 7 compares the memory consumption of the container orchestrator processes alone, without any other processes. In the case of Kubernetes, Flannel has been included because K3S and FLEDGE provide container networking by default. These results show that in its current state, FLEDGE uses only about 25% as much memory as Kubernetes, and 50% to 60% as much as K3S.

These results show that compared to both Kubernetes and K3S, FLEDGE uses significantly less resources, especially when comparing orchestrator processes directly.

7 Future Work

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

For starters, placing the Virtual Kubelets in the cloud may not be ideal. When running in the cloud, they can buffer commands in case a FLEDGE agent becomes unavailable, but they also require a small amount of storage and memory. Additionally, since all Virtual Kubelets are run in their own pod, the amount of master nodes in the cloud will have to scale with the maximum number of pods per node, instead of using one master node to manage all edge nodes. For these reasons it may be more efficient to integrate the Virtual Kubelet into the FLEDGE agent.

Many other container runtimes than the ones used in FLEDGE exist, including rkt [37] and CRI-O [38]. Docker and Containerd were chosen because of their popularity and support for container standards, but it is possible that other container runtimes use less resources.

Because K3S is based on Kubernetes, it may be possible to modify FLEDGE so that it can also connect to K3S clusters. Considering the original use of Virtual Kubelets, it could also pass Kubernetes deployments to K3S.

FLEDGE uses OpenVPN to build a network environment, but many other VPN solutions exist, which may prove to be faster or more reliable for use with FLEDGE.

8 Conclusion

The introduction puts forward three requirements for an effective container orchestrator on edge devices.

FLEDGE is presented as a solution that meets these requirements. A VPN is used to homogenize edge networks and to provide a basic layer of security for communication between the edge and the cloud. Compatibility with container standards is achieved by using OCI API's to communicate with container runtimes. CNI can be safely ignored using a custom implementation without affecting the rest of the cluster. Low resource requirements are achieved by choosing the optimal container runtime and through the custom implementation of select functionality, such as container networking.

To validate the low resource requirements of FLEDGE, a number of evaluations are performed. The resource requirements for FLEDGE using both Docker and Containerd are examined, showing that Containerd only needs about half the resources Docker does, and confirming that it is the optimal container runtime for FLEDGE.

K3S and Kubernetes are discussed as alternatives to FLEDGE, and evaluated to determine their resource requirements. The results shows that FLEDGE only requires 50–60% less resources than a Kubernetes worker node, and around 25–30% less resources than K3S on ARM devices. On x64, FLEDGE resource requirements are similar to those of K3S.

In conclusion, FLEDGE can deploy Kubernetes pods on edge devices while using significantly less resources than either Kubernetes or K3S. Despite this, it is highly experimental and many topics for future work on improving FLEDGE are discussed.

Acknowledgment. The research in this paper has been funded by Vlaio by means of the FLEXNET research project.

References

1. Felter, W., Ferreira, A., Rajamony, R., Rubio, J.: An updated performance comparison of virtual machines and Linux containers. In: 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS) (2015). https://doi.org/10.1109/ISPASS.2015.7095802

2. What is Kubernetes? https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/
3. Why Docker? https://www.docker.com/why-docker
4. CNM vs CNI. https://www.nuagenetworks.net/blog/container-networking-standards/
5. About OCI. https://www.opencontainers.org/about
6. Virtual kubelet. https://github.com/virtual-kubelet/virtual-kubelet
7. Mach, P., Becvar, Z.: Mobile edge computing: a survey on architecture and computation offloading. IEEE Commun. Surv. Tutor. **19**(3), 1628–1656 (2017). https://doi.org/10.1109/COMST.2017.2682318
8. Kumar, K., Lu, Y.-H.: Cloud computing for mobile users: can offloading computation save energy? Computer **43**, 51–56 (2010). https://doi.org/10.1109/MC.2010.98
9. Villari, M., Fazio, M., Dustdar, S., Rana, O., Ranjan, R.: Osmotic computing: a new paradigm for edge/cloud integration. IEEE Cloud Comput. **3**(6) (2016). https://doi.org/10.1109/MCC.2016.124
10. Santoro, D., Zozin, D., Pizzolli, D., De Pellegrini, F., Cretti, S.: Foggy: a platform for workload orchestration in a Fog Computing environment. In: 2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom) (2017). https://doi.org/10.1109/CloudCom.2017.62
11. Morshed, A., et al.: Deep osmosis: holistic distributed deep learning in osmotic computing. IEEE Cloud Comput. **4**(6) (2017). https://doi.org/10.1109/MCC.2018.1081070
12. Kubernetes federation. https://kubernetes.io/docs/concepts/cluster-administration/federation/
13. Wauters, T., et al.: Federation of internet experimentation facilities: architecture and implementation. In: Proceedings of the European Conference on Networks and Communications, pp. 1–5 (2014)
14. Moreno-Vozmediano, R., et al.: BEACON: a cloud network federation framework. In: Celesti, A., Leitner, P. (eds.) ESOCC Workshops 2015. CCIS, vol. 567, pp. 325–337. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-33313-7_25
15. Bottoni, P., Gabrielli, E., Gualandi, G., Mancini, L.V., Stolfi, F.: FedUp! Cloud federation as a service. In: Aiello, M., Johnsen, E.B., Dustdar, S., Georgievski, I. (eds.) ESOCC 2016. LNCS, vol. 9846, pp. 168–182. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-44482-6_11
16. Goethals, T., Kerkhove, D., Van Hoye, L., Sebrechts, M., De Turck, F., Volckaert, B.: FUSE: a microservice approach to cross-domain federation using docker containers. In: CLOSER 2019, the 9th International Conference on Cloud Computing and Services Science, pp. 90–99 (2019)
17. Puthal, D., Nepal, S., Ranjan, R., Chen, J.: Threats to networking cloud and edge datacenters in the internet of things. IEEE Cloud Comput. **3**(3) (2016). https://doi.org/10.1109/MCC.2016.63
18. Villari, M., Fazio, M., Dustdar, S., Rana, O., Chen, L., Ranjan, R.: Software defined membrane: policy-driven edge and internet of things security. IEEE Cloud Comput. 4(4) (2017). https://doi.org/10.1109/MCC.2017.3791014
19. Chowdhury, N.M.M.K., Boutaba, R.: Network virtualization: state of the art and research challenges. IEEE Commun. Mag. **47**(7) (2009). https://doi.org/10.1109/MCOM.2009.5183468
20. Hamed, H., Al-Shaer, E., Marrero, W.: Modeling and verification of IPSec and VPN security policies. In: 13th IEEE International Conference on Network Protocols (ICNP 2005) (2005). https://doi.org/10.1109/ICNP.2005.25

21. Pohl, F., Schotten, H.D.: Secure and scalable remote access tunnels for the IIoT: an assessment of openVPN and IPsec performance. In: De Paoli, F., Schulte, S., Broch Johnsen, E. (eds.) ESOCC 2017. LNCS, vol. 10465, pp. 83–90. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-67262-5_7
22. Kotuliak, I., Rybár, P., Trúchly, P.: Performance comparison of IPsec and TLS based VPN technologies. In: 2011 9th International Conference on Emerging eLearning Technologies and Applications (ICETA) (2011). https://doi.org/10.1109/ICETA.2011.6112567
23. Pahl, C., Lee, B.: Containers and clusters for edge cloud architectures - a technology review. In: 2015 3rd International Conference on Future Internet of Things and Cloud (2015). https://doi.org/10.1109/FiCloud.2015.35
24. Dupont, C., Giaffreda, R., Capra, L.: Edge computing in IoT context: horizontal and vertical Linux container migration. In: 2017 Global Internet of Things Summit (GIoTS) (2017). https://doi.org/10.1109/GIOTS.2017.8016218
25. Rancher Labs - K3S Lightweight Kubernetes. https://k3s.io/
26. Xiong, Y., Sun, Y., Xing, L., Huang, Y.: Extend cloud to edge with KubeEdge. In: 2018 IEEE/ACM Symposium on Edge Computing (SEC) (2018). https://doi.org/10.1109/SEC.2018.00048
27. Verma, A., Pedrosa, L., Korupolu, M., Oppenheime, D., Tune, E., Wilkes, J.: Large-scale cluster management at Google with Borg. In: EuroSys 2015 Proceedings of the Tenth European Conference on Computer Systems, Article No. 18 (2015)
28. KubeEdge: A Kubernetes Native Edge Computing Framework. https://kubeedge.io/en/
29. What is KubeEdge: Architecture. https://docs.kubeedge.io/en/latest/modules/kubeedge.html#architecture
30. Light, R.A.: Mosquitto: server and client implementation of the MQTT protocol. J. Open Source Softw. https://doi.org/10.21105/joss.00265
31. imec Virtual Wall. https://www.ugent.be/ea/idlab/en/research/research-infrastructure/virtual-wall.htm
32. The DF command. https://www.linuxjournal.com/article/2747
33. pmap - report memory map of a process. https://linux.die.net/man/1/pmap
34. Propertional Set Size (PSS). http://lkml.iu.edu/hypermail/linux/kernel/0708.1/3930.html
35. Docker components explained. http://alexander.holbreich.org/docker-components-explained/
36. kube-proxy. https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/
37. Getting started with rkt. https://coreos.com/rkt/docs/latest/getting-started-guide.html
38. CRI-O, lightweight container runtime for Kubernetes. https://cri-o.io/