Lightweight container alternatives for deploying Kubernetes workloads in the edge

Maxim De Clercq
Student number: 01803078

Supervisors: Prof. dr. ir. Filip De Turck, Prof. dr. Bruno Volckaert
Counsellors: Dr. ing. Tom Goethals, ing. Merlijn Sebrechts

Master’s dissertation submitted in order to obtain the academic degree of
Master of Science in Computer Science Engineering

Academic year 2022-2023
This master's dissertation concludes the end of my five-year study programme at Ghent University. Choosing this subject was an easy decision due to my inherent affinity towards kernels and containerization. After broadening my knowledge about the internals of container images, I do not think that I will ever be able to use Docker in the same way as before.

The successful completion of this work would not have been possible without the valuable counseling from Tom Goethals, who has dedicated countless hours to helping me understand complex subjects and providing invaluable feedback. I would also like to express my appreciation towards Filip De Turck, Bruno Volckaert, and Merlijn Sebrechts, whose valuable suggestions and insights have brought this master's dissertation to a higher level.

By finishing this master's dissertation, my studies come to a close, which have been nothing short of a journey. The Vlaamse Technische Kring Gent and the FaculteitenKonvent Gent have been my second family, without whom I would never have met the most meaningful people in my life, nor would I have become the person I am today. Throughout the hard times, especially during the COVID-19 pandemic, my friends and my dog Otto have always supported me, for which I am forever grateful.

Lastly, I want to thank my parents, my sister, but also my girlfriend Elizabeth-Rebecca, for supporting me during the writing of this master's dissertation. However, I do believe that there are not enough ways to ever thank my parents for their support during the challenging periods throughout my education, especially during high school and the COVID-19 pandemic. Their love, nurturing and support serve as the bedrock upon which I could bring my studies to completion, for which I can be nothing but grateful.

Maxim De Clercq
June 2023
Permission of use on loan

The author gives permission to make this master’s dissertation available for consultation and to copy parts of this master’s dissertation for personal use. In all cases of other use, the copyright terms have to be respected, in particular with regard to the obligation to state explicitly the source when quoting results from this master’s dissertation.

Maxim De Clercq
June 2023
Remark on the master’s dissertation

This master’s dissertation is part of an exam. Any comments formulated by the assessment committee during the oral presentation of the master’s dissertation are not included in this text.
Abstract

Recently, different paradigms for the efficient execution of edge workloads have become popular. Osmotic computing is a distributed computing paradigm that allows seamless integration of cloud and edge resources through the use of fluid migration of data and computation across multiple devices and networks. By moving computation close to where the relevant data is collected, resources in the edge can be exploited to support large scale deployments. Orchestrators such as Kubernetes are useful for edge computing, because they provide an infrastructure to manage and orchestrate containerized workloads across distributed computing environments, enabling efficient and flexible utilization of cloud and edge resources. However, they are resource intensive, even when idle. This is undesirable in the edge because these devices often have very little resources. As a result, it is still not very practical to use container orchestrators for osmotic computing.

To be able to use Kubernetes on edge devices, its memory footprint has to be reduced drastically. Fortunately, there are various ways to optimize the memory usage. Frameworks like ioFog, KubeEdge and FLEDGE do exactly this. However, every implementation currently makes use of container technologies like containerd, CRI-O and Docker.

This master’s dissertation proposes a Kubernetes-compatible solution that extends Kubernetes beyond container runtimes. By employing unikernel technologies such as OSv, it is able to execute applications more efficiently in the edge, resulting in a memory reduction of up to 20% for Java applications. The proposed solution, named Feather, enables the deployment of these unikernels in existing Kubernetes clusters, only consuming 98 MiB of extra memory on the node.
Lightweight container alternatives for deploying Kubernetes workloads in the edge

Maxim De Clercq

Ghent University

DeClercq.Maxim@UGent.be

Abstract—Recently, different paradigms for the efficient execution of edge workloads have become popular. Orchestrators such as Kubernetes are useful for edge computing, providing an infrastructure to efficiently manage and orchestrate containerized workloads across distributed computing environments. However, they are very resource intensive, even when idle. In the edge, this is undesirable because of the limited resources available. While several Kubernetes distributions have been developed in order to reduce this resource overhead, they limit themselves to container runtimes. This paper proposes a Kubernetes-compatible solution that extends Kubernetes beyond container runtimes. By employing unikernel technologies such as OSv, it is able to execute applications more efficiently in the edge, resulting in a memory reduction of up to 20% for Java applications. The proposed solution, named Feather, enables the deployment of these unikernels in existing Kubernetes clusters, only consuming 98 MiB of extra memory on the node.

Keywords—edge, kubernetes, container, microvm, unikernel

I. INTRODUCTION

With the increasing popularity of Internet of Things (IoT), it is incorporated in even more daily-used devices, from specialized medical appliances, to smart fridges. IoT Analytics forecasts that the number of connected IoT devices will surpass 29 billion in 2027 [1]. The data stream generated by these devices will be massive, and the existing cloud infrastructure has insufficient capacity to process all this data.

The concept of edge computing brings computing and storage closer to where the relevant data is produced. As a result, it is able to reduce latency and improve throughput.

A. Problem

Edge devices are becoming increasingly more powerful, encouraging the shift to process more data on the edge device itself. They have even become powerful enough to run containerized workloads, enabling the use of orchestrators such as Kubernetes [2]. Unfortunately, Kubernetes agents add a significant resource overhead, rendering them unsuitable for edge devices as is.

Recent developments such as ioFog [3], KubeEdge [4] and FLEDGE [5] provide alternative implementations that reduce the overhead as much as possible. However, besides still adding some overhead, the main limitation of these implementations is that they are based on containerized workloads.

B. Goal

This paper proposes an alternative Kubernetes-compatible solution based on FLEDGE [5], which is not limited to the use of containers, enabling the deployment of workloads based on other technologies such as MicroVMs [6]. Consequently, it facilitates the evaluation of possible benefits of using non-container workloads in Kubernetes clusters.

Unfortunately, these technologies cannot interface with the Kubernetes ecosystem, making it more difficult to incorporate alternatives (e.g. MicroVMs) into an existing workflow [7]. This paper presents a complete solution for packaging, distributing and deploying workloads in a runtime-agnostic manner. It aims to create a deployment workflow nearly identical to container management, to allow seamless integration with the Kubernetes ecosystem.

II. EDGE TECHNOLOGIES

A. Virtualization

Both containers and MicroVMs are examples of degrees of virtualization, where at least some degree of isolation from the host system is established. Originally defined by Popek and Goldberg as “an efficient, isolated duplicate of a real computer machine”, virtual machines (VMs) act as a full physical machine and provide the functionality needed to execute entire operating systems [8].

Traditional VMs often run full operating systems, which can run several unmodified individual applications. Micro virtual machines (MicroVMs) are a lightweight form of virtualization designed to run individual workloads or applications. There are several technologies that enable the creation of MicroVMs, but library OSes are the most flexible [9]. Typical services provided by the operating system such as networking, disk management, etc., are provided in the form of libraries and composed with the application and its configuration to construct a VM image [9]. Unikernels are a form of library OS, consisting of a specialized set of libraries that contain the minimal set of functionality in order to run the application [10].
These libraries are then compiled together with the application to build a self-contained image (a unikernel) which is run directly on a hypervisor or hardware without an intermediate host kernel such as Linux \(^1\). As a result, unikernels have minimal overhead and reduce the amount of context switches between kernel- and user-space to a minimum.

In contrast, containerization is a technique which isolates processes by making use of Linux kernel namespaces and cgroups \([11]\). Because containerization uses features of the host Linux kernel instead of introducing a hypervisor as an intermediate layer, containers tend to start significantly faster than virtual machines \([12]\). A core concept of containers is that they all share the same Linux host kernel. As a result, containers are less secure, but generally more resource efficient \([13]\).

**B. Orchestration**

Orchestration allows the provisioning of resources such as virtual machines and containers over a large number of devices in the cloud. Orchestrators offer an extensive solution for infrastructure management, including networking and storage features. Kubernetes, a popular open-source platform, offers a powerful solution for deploying containerized applications \([2]\). A popular Kubernetes distribution specifically designed for the edge is KubeEdge. KubeEdge is able to achieve a memory overhead of only 70 MiB, although it does require modifications to its master nodes, which is suboptimal \([4]\). IoFog, another popular orchestration platform for the edge, has a memory footprint of only 100 MiB on worker nodes, which is fairly little in comparison with some other Kubernetes distributions \([3]\).

A important component of all orchestrators is the agent (e.g. kubelet for Kubernetes), which is responsible for communicating to the API server of the control-plane. It manages containers on worker nodes and reports their statuses back to the control-plane. An adapter for the Kubernetes kubelet has been created, called the virtual kubelet \([14]\). The virtual kubelet intercepts commands from the control-plane and passes them on to one of its providers, which enables the creation of custom kubelet implementations.

**C. FLEDGE**

A recent virtual kubelet implementation is FLEDGE, which is designed for minimal resource overhead \([5]\). This Kubernetes-compatible node agent is created with three design goals in mind.

1) Compatibility with modern standards for container orchestration or providing an adequate alternative.
2) Provide secure communications between edge devices and the cloud by default, with minimal impact on local networks.
3) Have low resource requirements, primarily in terms of memory, but also in terms of processing power and storage.

---

\(^1\) In case of a type 1 hypervisor.

---

![Fig. 1: High-level comparison between the components of FLEDGE and Feather.](image)

By directly receiving commands from the control-plane, FLEDGE is able to have full control over the actions that are performed in reaction to these commands. These commands are received by a provider, which is called the broker. This broker passes these commands to a Container Runtime Interface (CRI) implementation that is able to communicate with a container runtime. A CRI is the component that communicates with the runtime, and as such is responsible for managing the containers themselves. FLEDGE currently supports both containerd and Docker as a runtime, but is limited to the use of containers.

**III. Architecture**

**A. Deployment**

Building upon FLEDGE, the goal is not only to explore lightweight containerization alternatives, but also to provide a full packaging and deployment solution for these alternatives. To avoid confusion with FLEDGE, the new solution is named Feather. Feather makes use of the same virtual kubelet pattern, but because it is not limited to containers, the CRI used in FLEDGE is referred to as a “backend”. For the same reason, workloads inside pods are referred to as “instances”.

Both FLEDGE and Feather share a very similar high-level design as shown in Fig. 1. The FLEDGE broker is responsible for setting up the node and passing API server commands to a runtime interface. The runtime interfaces in FLEDGE are responsible for managing the entire workload, from networking to volume management. Feather differentiates itself by incorporating as much pod logic as possible into the provider, which makes it more complicated than the FLEDGE broker. This means that it is not required for a backend to have all the knowledge about the pod itself, since they manage instances independently from each other. As a result, the complexity of Feather backend implementations is reduced.

The provider extracts the relevant information from the `core/v1.PodSpec` (e.g. networking, volumes) on a per-instance basis and embeds it into an extended
core/v1.Container specification. For example, if the pod is globally configured to use host networking, the backend still needs that information to create its workloads under that condition. As a result, the backend only needs to receive the extended core/v1.Container specification in order to set up the correct configuration for an instance. The backend is then only responsible for pulling images for instances and managing their life-cycles.

Feather is designed to be generic, such that the specific nature of such a runtime is not relevant. This means that Feather is able to support anything from container runtimes to hypervisors. In order to function as a sufficient proof-of-concept of this design, Feather supports both containers and MicroVMs through containerd and OSv respectively.

For compatibility purposes, the backend design is inspired by the OCI Runtime Spec [15]. This specification describes the configuration, execution environment, and life-cycle of a container. Since containerd is a native container runtime supported by default by Kubernetes, and because FLEDGE has a CRI for it, creating a backend for it is self-evident. Implementing a backend for OSv is not as straightforward. First of all, instead of with a container runtime, the backend needs to communicate with a hypervisor. Consequently, the backend needs to extract a VM disk image from the OCI Image Spec compliant image. The backend then needs to determine which parameters need to be passed to the hypervisor. Besides increased complexity, this can introduce compatibility issues (e.g. for persistent storage and mounts).

B. Packaging

Packaging non-container images for backend-agnostic storage is a non-trivial problem. For example, OSv images and other MicroVMs are packaged as raw disk images without any additional information [16]. In order to distribute these atypical “container” images reliably and correctly, well-established container image standards are used in combination with minimal metadata.

The most straightforward solution is to store a disk image as a layer in an OCI Image, which enables configuration similar to container images. In order to determine the content type of an OCI Image Layer, the OCI Image Config is extended, allowing Feather to select the correct backend for an image.

1) feather.backend: The backend that this image was designed for.
2) feather.runtime (optional): The preferred runtime to schedule this instance with. In the case of virtual machines, this refers to the used hypervisor.

Since containerd images are already OCI Images, these fields are only required for the OSv backend. Whenever the feather.backend field is not present, Feather can safely assume a container image.

IV. EVALUATION

When determining alternative runtimes, the most important aspect for edge computing is reducing resource overhead. In order to assess the performance and capabilities of the two Feather backends ‘containerd’ and ‘OSv’, three different scenarios are analyzed.

1) Baseline. Without any instances running, does this backend make use of any resources?
2) Minimal. Does the backend impose a significant overhead when the instance is doing absolutely nothing?
3) Application benchmark. How well does the backend perform when it is used for a practical application?

For the purpose of the experiment, a master node and a worker node are set up, both running Ubuntu 20.04.05 LTS. Both have a 2x Quad core (2.2GHz) CPU, 12 GB of RAM, 160 GB of storage, and two 1 Gb/s network interfaces. In order to emulate an edge device with only 1 CPU and 1 Gb of RAM, Feather limits containerd and OSv with cgroups and QEMU options respectively.

A. Baseline

To measure the overhead of Feather and the services required by the backends, a scenario without any instances running is analyzed.

The Feather binary used for the experiment is a stripped, statically linked binary that is built for the x86_64 architecture. This binary is 53.56 MiB in size, around 15 MiB larger than the FLEDGE binary. This is caused by both supporting more recent versions of Kubernetes and having additional support for OSv images. In comparison, a single KubeEdge v1.13.0 agent is 68.99 MiB in size.

Both backends have runtime dependencies, such as the containerd daemon itself and a hypervisor. Installing containerd on Ubuntu 20.04.05 LTS requires about 116 MiB, which is less than installing the QEMU hypervisor and its dependencies, which require 376 MiB. A pre-built QEMU hypervisor package depends on many components that provide a wide range of features, such as UI drivers. QEMU can be built from source to reduce this storage overhead, excluding all features that are non-essential for the OSv backend.

![Fig. 2: CPU usage of relevant Feather processes in the baseline scenario.](image-url)

The only relevant processes consuming resources on the worker node are feather and containerd. Fig. 2 shows that the CPU usage of feather has an initial peak at around 23%,
dropping to an average of 1.7% when idle. As shown in Fig. 3, Feather uses an average of 98 MiB of memory, which seems to be significantly more than FLEDGE, which is able to achieve an average of 50 MiB while idling on x86_64 architectures [5]. The containerd process has no significant CPU usage, and uses 55 MiB of memory on average. Neither qemu-system-x86_64 or virtiofsd are running if there is no managed instance present, so in this case the OSv backend is not consuming any resources.

The scenario for analyzing a minimal workload makes use of a simple Hello World application written in C, which sleeps for the duration of the experiment. This ensures that there is no work done, but that it is still possible to verify if the application has started successfully.

It is important to note that the minimal scenario is run once for every backend, because it is useful to analyze the resources used by Feather in both cases. Fig. 4 shows that containerd uses some extra CPU at container creation, while OSv does not. This is likely because the containerd backend involves complex communication with an external daemon, whilst the OSv backend only needs to prepare the configuration and disk files for the instance. In both scenarios, there is only some additional memory usage during pod creation at 0:30.

Both containerd and containerd-shim-runc-v2 use an insignificant amount of processing resources, except at container creation, when containerd uses 4.2% of the CPU. Containerd uses 56.45 MiB of memory on average, while containerd-shim-runc-v2 uses around 9.46 MiB. This results in a total memory overhead of 65.76 MiB on average.

The QEMU hypervisor directly uses a significant amount of processing power, idling at 11.5%. With a memory consumption of 63.28 MiB on average, QEMU does perform slightly better than containerd and containerd-shim-runc-v2 combined.

Note that there are no metrics for virtiofsd included in Fig. 4. This is because Feather did not start virtiofsd, as the minimal scenario does not make use of any volume mounts.

In order to be able to analyze and evaluate the features of Feather, it is necessary to determine how well Feather and its backends perform under the load of a resource-intensive application. The deployment of a Minecraft game server [17], running on a Java Virtual Machine (JVM), has been selected as the ideal real-world scenario. Since the edge device only has 1 GiB of free memory, it is a reasonable choice to only allocate a maximum of 800 MiB of memory to the Java application, leaving 200 MiB of memory for the operating system, Feather, and its backends. Unlike the minimal application, this application can be configured through a ConfigMap. A Minecraft game server is initialized with a seed, where different seeds lead to different worlds to be generated. By keeping the seed consistent between backends, a deterministic testing environment can be created. Amongst the seed, other properties like the default game mode can be configured. For this experiment, the Minecraft worlds are initialized with seed -11970185.
As with the minimal scenario, the application scenario is performed once for every backend. To ensure representative result the tests are performed as follows.

1) Any previous deployment is deleted.
2) The server image is deployed on the worker node using Kubernetes. The logs then state the startup time of this server, and verify the configuration of the seed through the ConfigMap.
3) After 2 minutes, 4 clients join the server at the same time. This verifies the ability of the server to handle simultaneous events, such as chunk generation, which stresses the CPU.
4) After 4 minutes, villager entities are spawned at a rate of about 3 villagers per 10 in-game ticks. On a server that is fully responsive at 20 in-game ticks per second, this corresponds to a rate of 6 villagers per second.

The course of the tests is shown in Fig. 6. The responsiveness of the server (expressed in ticks per second, or TPS) drops slightly after 2 minutes, and drops steadily after 4 minutes. The rate at which villagers are spawned also decreases in time, which notes its relation to the TPS as mentioned in Step 4.

![Fig. 6: Ticks per second and number of villagers in the application scenario.](image)

As interpreted from the logs of the instances, the containerd instance starts in 38.713s, while the OSv instance takes 37.199s to start, which is 18% faster. It is noticeable in Fig. 6 that the TPS drops slightly after 2 minutes, and drops steadily after 4 minutes. The rate at which villagers are spawned also decreases in time, which notes its relation to the TPS as mentioned in Step 4.

By the end of the test, Fig. 7 shows that qemu-system-x86_64 uses only 661.10 MiB of RAM, which is remarkably less than its container counter-part, which is using 820.57 MiB of memory at that time. When looking at JVM statistics, Fig. 9 shows that the OSv image spends a lot less time on garbage collection, and Fig. 10 shows that it only uses a minimal amount of allocated memory. The containerd instance however, allocates significantly more memory and has a more jagged “used memory” curve. This result is expected, since OSv claims that it is able to expose extra hardware features to the JVM because it is running in the same address space. This allows the JVM to manage memory much more efficiently, resulting in less allocations and time spent on garbage collection, as shown in Fig. 8 and Fig. 9.

![Fig. 7: CPU usage of relevant Feather processes in the application scenario.](image)

![Fig. 8: Resident memory usage of relevant Feather processes in the application scenario.](image)
D. Analysis

1) Non-functional Requirements: This evaluation is performed for a reference edge device that has 1 CPU and 1 GiB of memory. Evidently, QEMU does seem to use more CPU than the CPU limits that are set by Kubernetes. QEMU also introduces some CPU overhead that is not attributed to the application as seen in the minimal scenario. This means that Feather is required to implement cgroup management for the OSv backend in order to have a true limitation of the CPU resources that QEMU is able to consume. An alternative solution is to use a different hypervisor than QEMU which does not have this issue.

In terms of memory, QEMU far outperforms the containerized Java application. The OSv instance requires 20% less memory than the containerd instance. This improved memory management is also clearly visible in the increased “used memory” stability of the JVM.

E. Functional Requirements

It is shown that Feather enables the configuration and deployment of both containerd and OSv images with Kubernetes. Contrary to the container backend, not everything is compatible with the OSv backend, due to hypervisor or OSv limitations.

In Kubernetes environments, it is a common practice to create PersistentVolume objects in order to store data persistently. Feather has no support for PersistentVolumes, which is a limitation of the Virtual Kubelet itself.

In addition, QEMU is only able to create bind mounts for directories through virtio-fs, but this is not possible for files. In addition, the virtio-fs driver provided by the OSv kernel only supports read-only mounts. This means that it is impossible to use any bind mount for persistent storage in the OSv backend. The only filesystems that also have drivers, and can be used for persistent storage, are NFS and ZFS. Since Kubernetes only supports NFS filesystems as a VolumeSource, this seems like the only option to store files persistently for the OSv backend.

V. Future Work

In order to improve Feather, it is possible to develop backends for other technologies, such as WASM, Kata Containers, Firecracker or Unikraft. In addition, Feather does not currently support advanced networking features and capabilities in the edge. Only providing minimal internet access and application-level NAT support, some work is required to fully integrate Feather OSv instances into a container network.

VI. Conclusion

Feather provides a solution for extending the Kubernetes ecosystem with non-container alternatives such as unikernels, which can result in a reduced resource overhead. By leveraging existing standards such as the OCI Specifications and the Kubernetes ecosystem, this master’s dissertation ensures that the proposed solution is seamlessly interoperable with the current infrastructure, requiring minimal effort by the developer to leverage the capabilities of Feather.

Feather has made great improvements upon FLEDGE, extending this Kubernetes agent to non-container runtimes, such as OSv. This work opens up possibilities for further research and development, including the research towards more lightweight container alternatives for deploying Kubernetes workloads in the edge.

REFERENCES


## Contents

**Abstract**  

**List of Figures** xvi  

**List of Tables** xvii  

**List of Acronyms** xviii  

**List of Code Fragments** xix  

1 Introduction  
1.1 Problem  
1.2 Goal  

2 Edge Technologies  
2.1 Virtualization  
2.1.1 Virtual Machines  
2.1.2 MicroVMs  
2.1.3 Containerization  
2.2 Orchestration  
2.2.1 Kubernetes  
2.2.2 Kubernetes Distributions  
2.2.3 Kubelet  
2.3 FLEDGE  
2.3.1 Runtimes  

3 Architecture  
3.1 Deployment  
3.1.1 Virtual kubelef  
3.1.2 Backend  
3.2 Packaging  
3.2.1 Concept  
3.2.2 Backend  

v

xvi

xvii

xviii

xix

1

1

1

3

3

4

4

5

6

8

9

10

12

13

13

15

15

16
List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>High-level comparison of virtual machines, microVMs and containers</td>
<td>5</td>
</tr>
<tr>
<td>2.2</td>
<td>The different components of a Kubernetes cluster</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>The usage of a virtual kubelet inside an existing Kubernetes cluster</td>
<td>9</td>
</tr>
<tr>
<td>2.4</td>
<td>Architectural overview of the broker pattern used in FLEDGE and its use of the virtual kubelet [1]</td>
<td>10</td>
</tr>
<tr>
<td>3.1</td>
<td>High-level comparison between the components of FLEDGE and Feather</td>
<td>13</td>
</tr>
<tr>
<td>3.2</td>
<td>Architectural overview of deploying an application with Feather</td>
<td>18</td>
</tr>
<tr>
<td>5.1</td>
<td>CPU and memory usage of relevant Feather processes in the baseline scenario</td>
<td>38</td>
</tr>
<tr>
<td>5.2</td>
<td>CPU and memory usage of relevant Feather processes in the minimal scenario</td>
<td>39</td>
</tr>
<tr>
<td>5.3</td>
<td>Ticks per second in comparison with the number of villagers in the application scenario</td>
<td>41</td>
</tr>
<tr>
<td>5.4</td>
<td>CPU and memory usage of relevant Feather processes in the application scenario</td>
<td>41</td>
</tr>
<tr>
<td>5.5</td>
<td>Fraction of CPU time spend on garbage collection by the JVM in the application scenario</td>
<td>42</td>
</tr>
<tr>
<td>5.6</td>
<td>Used and allocated JVM memory statistics in the application scenario</td>
<td>42</td>
</tr>
</tbody>
</table>
List of Tables

3.1 The interface of the provider in Feather ........................................ 14
3.2 The interface of a backend in Feather ........................................ 14
4.1 Description of the relevant fields in a Capstan configuration ............... 26
4.1 Technical explanation of the recommended JVM flags for a PaperMC server ........................................ 32
5.2 Minecraft seeds that are used for initializing the worlds ...................... 32
5.3 Packages with a significant contribution to FLEDGE’s and Feather’s binary sizes (>1.0 MiB) .............. 37
## List of Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
<th>Pages</th>
</tr>
</thead>
<tbody>
<tr>
<td>API</td>
<td>application programming interface</td>
<td>8, 10, 21, 24, 25, 27, 33, 35, 36, 39</td>
</tr>
<tr>
<td>APT</td>
<td>advanced packaging tool</td>
<td>7, 8</td>
</tr>
<tr>
<td>AWS</td>
<td>Amazon Web Services</td>
<td>8</td>
</tr>
<tr>
<td>CNCF</td>
<td>Cloud Native Computing Foundation</td>
<td>8</td>
</tr>
<tr>
<td>CRI</td>
<td>Container Runtime Interface</td>
<td>8-12</td>
</tr>
<tr>
<td>DNS</td>
<td>Domain Name System</td>
<td>28</td>
</tr>
<tr>
<td>GC</td>
<td>garbage collection</td>
<td>42</td>
</tr>
<tr>
<td>IO</td>
<td>input/output</td>
<td>26</td>
</tr>
<tr>
<td>JVM</td>
<td>Java virtual machine</td>
<td>xvii, 1, 5, 22, 23, 30, 31, 33</td>
</tr>
<tr>
<td>KVM</td>
<td>Kernel-based Virtual Machine</td>
<td>3, 26</td>
</tr>
<tr>
<td>MAC</td>
<td>media access control</td>
<td>26</td>
</tr>
<tr>
<td>microVM</td>
<td>micro virtual machine</td>
<td>xiii, xvi, 3, 5, 11, 13, 28, 39, 54, 55</td>
</tr>
<tr>
<td>NAT</td>
<td>network address translation</td>
<td>26, 45</td>
</tr>
<tr>
<td>NFS</td>
<td>Network File System</td>
<td>25, 44</td>
</tr>
<tr>
<td>OCI</td>
<td>Open Container Initiative</td>
<td>xix, 3, 10, 11, 15, 22, 24, 25, 27, 33, 35, 46, 56, 58</td>
</tr>
<tr>
<td>OS</td>
<td>operating system</td>
<td>3, 4</td>
</tr>
<tr>
<td>QEMU</td>
<td>Quick Emulator</td>
<td>3, 15, 28, 39, 42, 44, 45</td>
</tr>
<tr>
<td>RBAC</td>
<td>role-based access control</td>
<td>5</td>
</tr>
<tr>
<td>ROFS</td>
<td>Read-Only File System</td>
<td>25</td>
</tr>
<tr>
<td>TLS</td>
<td>Transport Layer Security</td>
<td>5</td>
</tr>
<tr>
<td>TPS</td>
<td>ticks per second</td>
<td>40, 42</td>
</tr>
<tr>
<td>VM</td>
<td>virtual machine</td>
<td>xvi, 3, 5, 11</td>
</tr>
</tbody>
</table>
List of Code Fragments

5.1 Extensions for Feather in an example OCI Image Configuration [2] .................................................. 17
4.1 Example of pushing a containerd image to an OCI Image Registry .................................................. 20
4.2 Example of pushing an OSv image to an OCI Image Registry ......................................................... 21
4.3 Example configuration for the Feather provider .................................................................................. 21
4.4 Extended Container structure, as consumed by a Feather backend .................................................. 22
4.5 Extended VolumeMount structure, as consumed by a Feather backend ............................................. 23
4.6 Extended VolumeMount structure, as consumed by a Feather backend ............................................. 23
5.1 Source code of the minimal application written in C ........................................................................ 30
5.2 ConfigMap definition for configuring the PaperMC application ....................................................... 33
5.3 Missing shell executable reported by the OSv instance in the application scenario (seed 1) .......... 43
5.4 Missing symbols reported by the OSv instance in the application scenario (seed 1) ..................... 43
A.1 Dockerfile for the minimal application ............................................................................................... 51
A.2 Capstan meta/package.yaml for the minimal application ................................................................ 51
A.3 Capstan meta/run.yaml for the minimal application .......................................................................... 51
A.4 Deployment definition for the minimal application ........................................................................... 52
A.5 Dockerfile for the PaperMC application ............................................................................................ 53
A.6 Capstan meta/package.yaml for the PaperMC application ............................................................... 53
A.7 Capstan meta/run.yaml for the PaperMC application ....................................................................... 54
A.8 Deployment definition for the PaperMC application ........................................................................... 55
1

Introduction

With the increasing popularity of Internet of Things (IoT), it is incorporated in even more daily-used devices, from specialized medical appliances, to smart fridges. IoT Analytics forecasts that the number of connected IoT devices will surpass 29 billion in 2027 \[^3\]. The data stream generated by these devices will be massive, and the existing cloud infrastructure has insufficient capacity to process all this data.

In the world of distributed computing \[^1\], edge computing brings computing and storage closer to where the relevant data is produced. As a result, it is able to reduce latency and improve throughput. Distributing workloads between the cloud and the edge has already been extensively researched, resulting into concepts such as osmotic computing \[^5\] and cloud offloading \[^3\].

1.1 Problem

Edge devices are becoming more and more powerful, encouraging the shift to process more data on the edge device itself. They have even become powerful enough to run containerized workloads, as opposed to bare-metal solutions or running applications directly on the host operating system. This sparks the interest if it is possible to use container orchestrators in order to optimally distribute these workloads over the edge devices. Popular container orchestrators like Kubernetes \[^7\] and HashiCorp Nomad \[^8\] have been around for quite some time. These container orchestrators have been designed with cloud computing in mind. Because the agents they run on edge devices add a significant overhead, they are currently not suited for running on edge devices.

There have been some recent developments towards Kubernetes agents that have been tailored towards the edge, like ioFog \[^9\], KubeEdge \[^10\] and FLEDGE \[^1\]. However, these agents still incur a significant resource overhead. Furthermore, the main limitation of these implementations is that they are based on containerized workloads.

1.2 Goal

This master’s dissertation explores the options of developing a customized Kubernetes agent which is not limited to the use of containers, enabling the deployment of workloads based on other technologies such as MicroVMs \[^11\]. Consequently, it

\[^1\] A system that consists of multiple software components that are on multiple computers, but run as a single system \[^4\].
1 Introduction

facilitates the evaluation of possible benefits of using non-container workloads in Kubernetes clusters. Unfortunately, these technologies cannot interface with the Kubernetes ecosystem, making it more difficult to incorporate alternatives (e.g. MicroVMs) into an existing workflow \[12\]. This master’s dissertation presents a complete solution for the packaging, distribution and deployment of workloads in a runtime-agnostic way. It aims to create a workflow nearly identical to container management, to allow seamless integration with Kubernetes.
2

Edge Technologies

Edge computing aims to bring computation closer to where the data is generated, reducing latency for end users and enabling real-time processing. Several technologies have been developed in order to establish this approach. Besides covering these technologies, this chapter also delves into broader concepts such as different types of virtualization. Amongst those virtual machines and containerization are currently the two most frequently used mechanisms to host applications [13].

First, different types of virtualization are discussed that are relevant for this master’s dissertation. Then, Section 2.2 discusses container-based orchestrators. Finally, the project that this master’s dissertation builds upon is discussed in Section 2.3.

2.1 Virtualization

Virtualization enables the creation of virtual environments within a single edge device. In literature, virtualization often refers to the concept of virtual machines, where each virtual machine is running its own operating system [14]. In the sense of this master’s dissertation, virtualization is referred to as a spectrum. Different degrees of virtualization exist, ranging from isolating full operating systems, to sandboxing single applications.

2.1.1 Virtual Machines

In the sense of virtualization, it is possible to refer to virtual machines as the traditional form of virtualization. Originally defined by Popek and Goldberg as “an efficient, isolated duplicate of a real computer machine”, VMs act as a full physical machine and provide the functionality needed to execute entire operating systems [14].

The key component of virtual machines is the hypervisor, which manages these virtual environments. Hypervisors provide a layer of abstraction for the underlying hardware resources, creating environments that replicate the behavior of physical devices. This enables the host operating system to run anything that would normally be able to run on physical hardware, like other operating systems. A type 1 hypervisor acts as a middle layer, interacting directly with the hardware like CPU, memory, disks, and networking, whilst a type 2 hypervisor runs on top of an existing operating system [15]. A commonly used type 1 hypervisor is KVM, which is an open-source virtualization module in the Linux kernel that allows the kernel to function as a hypervisor [16]. KVM is often used in conjunction with QEMU to provide additional management capabilities for virtual machines. The generalized structure of a virtual machine running on such a type 1 hypervisor can be seen in Figure 2.1.

---

1 The practice of isolating the execution environment of an application in order to increase the security on the host system.
In edge computing, resource optimization and isolation often play a significant role. With the use of multiple virtual instances, it is possible to effectively distribute the available resources between isolated environments. However, managing multiple operating systems on a single device adds a lot of overhead. That is where microVMs come into play.

2.1.2 MicroVMs

As mentioned in Subsection 2.1.1, traditional virtual machines (VMs) often run full operating systems, which can run several unmodified individual applications. However, this is not the only type of virtual machines around. Micro virtual machines (microVMs) are a lightweight form of virtualization designed to run individual workloads or applications. They work exactly like traditional virtual machines, but instead of incorporating a full operating system, they only include everything that is necessary to run the single application. This makes them more lightweight, minimizing the overhead and maximizing performance.

There are several technologies that enable the creation of microVMs, but library OSes are the most flexible \[17\]. Typical services provided by the operating system such as networking, disk management, etc., are provided in the form of libraries and composed with the application and its configuration to construct a VM image \[17\]. Unikernels are a form of library OS consisting of a specialized set of libraries that contain the minimal set of functionality in order to run the application \[18\]. These libraries are then compiled with the application to build a self-contained image (a unikernel) which is run directly on a hypervisor or hardware without an intermediate host kernel such as Linux \[2\], which can be seen in Figure 2.1b. As a result, unikernels have minimal overhead and reduce the amount of context switches between kernel- and user-space to a minimum.

2.1.3 Containerization

Containerization is a technique which, in contrast to virtual machines, isolates processes by making use of Linux kernel namespaces and cgroups \[19\]. Because containerization uses features of the host Linux kernel instead of introducing a hypervisor as an intermediate layer, containers tend to start significantly faster than virtual machines \[20\].

A core concept of containers is that they all share the same Linux host kernel, in contrast to virtual machines. Containers package the application and all its dependencies, while they share the Linux host kernel with other containers, as shown in Figure 2.1c. As a result, containers are less secure, but generally more resource efficient than VMs \[21\]. This is particularly useful in edge devices because of their limited available resources.

2.2 Orchestration

In the context of cloud deployments, orchestration is the process of automating and organizing the management of complex workloads. In this process, specialized toolkits called “orchestrators” stand at the center of this automation. Orchestration allows the provisioning of resources such as virtual machines and containers over a large number of devices in the cloud. Different orchestration platforms exist. The most popular platforms are Docker Swarm \[22\], Kubernetes \[7\] and HasiCorp Nomad \[8\]. These platforms can be used for the scalable and flexible deployment of workloads, in order to manage and

\[2\] In case of a type 1 hypervisor.
orchestrate diverse scenarios. They also offer an extensive solution for infrastructure management, including networking and storage features. On one hand there are services which allow developers to run long-lived applications such as web services. On the other hand there are tasks which account for short-lived jobs that have to be scheduled (such as cron jobs).

In the context of edge computing, orchestration can be leveraged in the same manner in order to distribute these workloads over the edge devices. This enables a resilient edge infrastructure, while also offering useful monitoring features. Since this master’s dissertation focusses on the capabilities of Kubernetes within edge deployments, it only discusses this platform.

### 2.2.1 Kubernetes

Kubernetes, a popular open-source platform, offers a powerful solution for deploying containerized applications. Containerized workloads are currently very flexible and widely supported, creating a familiar environment for building and deploying applications reliably. Kubernetes facilitates this even more by providing a wide range of orchestration functionalities.\(^7\) Kubernetes can be useful for edge computing service architectures due to a number of its features:

- **Standardization**: Edge devices are often very diverse. Kubernetes is a standardized platforms which implements numerous industry standards such as the OCI Specifications.

- **Orchestration**: Kubernetes is an orchestrator for containerized workloads, enabling the seamless deployment of applications on edge devices.

- **Self-healing**: Applications that fail or crash should normally be automatically restarted, replaced or rescheduled in case a node dies.

- **Automated rollouts and rollbacks**: Applications are seamlessly updated and monitored. If, for any reason, an update fails, the application is rolled back as needed.

- **Security policies**: Edge devices might be located in vulnerable or untrusted environments. Kubernetes offers robust administrative security features for communication and networking. This includes TLS, RBAC and network policies.\(^8\)

- **Extensibility**: Kubernetes offers extensibility in the form of operators that allow the developer to incorporate application-specific behavior and logic into the Kubernetes framework. Mostly designed for automation, the operator pattern

---

\(^7\) A Kubernetes specification that enables the control of network flow

\(^8\) A Kubernetes specification that enables the control of network flow
2 Edge Technologies

captures how code can be written in order to automate a task that is not included in the Kubernetes framework.

A Kubernetes environment that manages workloads is called a cluster. Such a cluster consists of a control plane and a set of worker nodes as can be seen in Figure 2.2. The control plane is responsible for managing the workloads and scheduling them on the worker nodes. In small Kubernetes environments, the control plane is generally managed by a single node, which is called the master node. It is possible to replicate the control plane over multiple master nodes for resilience.

![Figure 2.2: The different components of a Kubernetes cluster.](image)

In order to register itself and to communicate with the control plane, each node runs a kubelet, which is the primary "node agent". The kubelet is then responsible for managing the deployment of the workload on the worker node. Furthermore, every worker node also runs a kube-proxy instance, which is a network proxy that maintains the network rules on the worker node.

A Kubernetes workload is a deployed application that is running in one or more pods. A pod is the smallest and most basic unit of the Kubernetes system. It is a logical grouping of containers that are scheduled and run together. The simplifies the management and deployment of a single application. Applications can then be configured through the use of several other resources like config maps and secrets.

2.2.2 Kubernetes Distributions

Both cloud and edge service architecture deployments can have various specific requirements. As a result, different Kubernetes distributions have been developed. A Kubernetes distribution packs a modified version of the core Kubernetes components together with additional features and utilities tailored towards a specific use-case.

The fact that Kubernetes offers all these orchestration features, while also being designed for extensibility and scalability, is a double-edged sword. With a design aimed towards massive cloud deployments, it is able to provide robust and resilient deployments. However, it consumes a massive amount of resources in the process. Kubernetes runs a set of system daemons on its worker nodes, including the container runtime (e.g. containerd), kube-proxy, and the kubelet including cAdvisor. All of these daemons together add some fixed overhead to Kubernetes worker nodes [23].

6
2 Edge Technologies

Although there are already several Kubernetes distributions specifically designed for the edge, there are other Kubernetes distributions with relevant design features. This section discusses relevant distributions that are aimed towards reducing resource overhead.

Vanilla [7]

Kubernetes without any modifications is generally referred to as vanilla Kubernetes. In this form, Kubernetes provides the functionality for container orchestration it was designed for. This results in quite a significant overhead on the Kubernetes nodes. Generally, the resources for the system and Kubernetes are reserved by the kubelet itself, requiring up to 25% of the available memory for small nodes [24]. However, larger nodes that have more than 8 GiB of memory get away with reserving as little as 10% of their available memory. The larger the node, the more insignificant this overhead becomes, especially relative to the added functionality. Generally, for a vanilla deployment of Kubernetes, a minimum of 2 GiB of RAM is recommended [25].

Edge devices generally have very little available memory. An overhead of 25% on a device with only 1 GiB of available memory is simply not affordable, since this severely limits the amount of available memory for applications.

Clearly, vanilla Kubernetes was not designed with a low resource overhead in mind, therefore other distributions should be explored.

MicroK8s [26]

MicroK8s is a fairly complete Kubernetes solution designed to be lightweight, while still offering many features. It runs on as little as 540 MB of memory, but the system recommendations for workloads are still at least 20 GB of disk space and 4 GB of memory. While designed for lightweight single-node deployments for testing and development, it is clear that MicroK8s is not suited for any edge developments. However, MicroK8s is still very interesting due to its reduction in resources requirements.

K0s [27]

While being comparable to MicroK8s, K0s is a batteries-included Kubernetes distribution with the edge in mind. Also consuming about half a gigabyte of memory on worker nodes, it is not an ideal solution for very low-resource devices.

K3s [28]

K3s comes as a stripped-down distribution of Kubernetes. Only keeping the modern and essential features, it comes as a fully-featured distribution. The downside is that it is not compatible with a vanilla deployment. K3s requires a customized master node in order to function. While this particular distribution hasn't been researched a lot, it requires at least 256 MiB of memory, making it slightly more efficient than MicroK8s and K0s [29].

---

*Software that is bundled with a wide range of features and tools, such that it is self-contained and ready for immediate use.*
KubeEdge

KubeEdge, a relatively new project incubated by the CNCF, is able to achieve a memory footprint of only 70 MiB. KubeEdge also provides support for bidirectional communication, making it possible to communicate with edge nodes located in private subnet, which is often an issue in edge environments.

While this minimal resource usage is a drastic improvement over vanilla Kubernetes, it requires to set up extra KubeEdge control-plane components, limiting interoperability.

2.2.3 Kubelet

The kubelet is a critical component in the Kubernetes ecosystem. Upon starting, it registers the worker node with the API server and communicates its status. When the node is ready for accepting workloads, the control plane can start scheduling pods onto the newly registered worker node. The specifications of the pods are then communicated to the kubelet through the API server. Interacting directly with the container runtime, it is fully responsible for managing the lifetime of the container and communicating its status back to the API server.

Virtlet

A virtlet is a Kubernetes runtime plugin that allows running VM workloads as containers within a Kubernetes cluster. The virtlet manager implements the Kubernetes CRI interface and serves requests from the kubelet. To enable seamless integration with the Kubernetes ecosystem, these virtual machines are executed inside containers, together with a dedicated qemu-kvm instance.

Although the virtlet manager extends Kubernetes with the ability to execute VM workloads, it does so by adding a significant overhead. Not only are these VMs still containerized, but the virtlet depends on the kubelet for communicating with the Kubernetes API server. The advantage of using the virtlet manager is that it does provide support for VM workloads without making large modifications to either the kubelet or the node by making use of the Kubernetes CRI.

Virtual Kubelet

A virtual kubelet abstracts the kubelet by providing an adapter that can be used to implement a custom Kubernetes “node agent”. This custom agent is then able to receive commands from the API server like a kubelet would, but is able to fully manage the pods itself. This allows a developer to extend the capabilities of Kubernetes beyond the traditional concept of a kubelet that runs on a worker node.

This concept is generally used to extend Kubernetes to external compute and serverless platforms, such as Azure Container Instances and AWS Fargate. However, it can also be used to extend Kubernetes with non-traditional ways of running workloads, such as using systemd to start pods as service units. The use of a virtual kubelet inside an existing Kubernetes cluster is shown in Figure 2.3.
Recently, FLEDGE (Flexible virtualization in the Edge) has been developed by Tom Goethals [2]. FLEDGE is a virtual kubelet implementation designed for minimal resource overhead. This Kubernetes-compatible container orchestrator is created with three design goals in mind.

1. Compatibility with modern standards for container orchestration or providing an adequate alternative.

2. Provide secure communications between edge devices and the cloud by default, with minimal impact on local networks.

3. Have low resource requirements, primarily in terms of memory, but also in terms of processing power and storage.

This implementation is able to act as a Kubernetes-compatible container orchestrator with a mere overhead of only 60 MiB of memory. This low resource usage is achieved by certain design choices.

First of all, FLEDGE is a virtual kubelet implementation, which means that it receives and processes Kubernetes commands directly from the control-plane, enabling full control over what is happening on the edge device. Furthermore, FLEDGE implements a broker pattern. This broker processes the commands from the virtual kubelet and passes them on to an appropriate Container Runtime Interface (CRI), as seen in Figure 2.4. Note that this is a different kind of CRI from what has been seen in Subsection 2.2.1. These are referred to as a Kubernetes CRI and FLEDGE CRI in order to differentiate between them. The goal of the FLEDGE CRI is then to interpret the API server command, optionally set up networking, and offload the management of the container itself to a container runtime like containerd. As of now, FLEDGE only supports containerd and Docker as container runtimes. This means that there is still a lot left to explore.

This master’s dissertation builds upon FLEDGE by redefining the broker pattern in Chapter 3 and implementing an alternative FLEDGE CRI using OSv as a proof-of-concept.
2.3.1 Runtimes

A container runtime is a program that is responsible for managing the life-cycle of containers. FLEDGE communicates with the container runtime in order to create, start, stop and delete containers that are managed by it. Besides containerd and Docker, which are already supported in FLEDGE, there are many interesting runtimes still left to explore, which are not necessarily limited to container workloads.

### Containerd

Containerd is a performant container runtime that is often used in cloud environments, such as a Kubernetes cluster. It is well-established and follows popular standards such as the [OCI Image Spec](https://github.com/opencontainers/image-spec) and the [OCI Runtime Spec](https://github.com/opencontainers/runtime-spec). Containerd is a low-level runtime that is primarily focussed on managing the life-cycle of containers.

### Docker

Built on containerd, Docker is a more user-friendly container runtime that has a broader scope than containerd. Docker provides a higher-level API that takes care of logging, networking configurations and volume management. This simplifies the implementation of a FLEDGE CRI but introduces an additional overhead because of this abstraction. Docker also provides tooling for packaging [OCI](https://github.com/opencontainers/image-spec) images, which can subsequently be managed with containerd if desired.

### CRI-O

CRI-O, another container runtime, implements the Kubernetes CRI directly, without any additional layers or abstractions. By following a Kubernetes-native approach, CRI-O is able to achieve better performance than containerd in some scenarios. However, since FLEDGE already uses a custom implementation to communicate with container runtimes, there would not be a great benefit from supporting CRI-O.
2 Edge Technologies

**gVisor**

Besides container runtimes, many other virtualization techniques exist. Some of these techniques are aimed towards providing better isolation when running containers, such as gVisor. GVisor is an application kernel that implements a substantial portion of the Linux system call interface. It provides an additional layer of isolation between running applications and the host operating system. By doing so, it is able to create an isolation layer between the Linux host kernel and container runtimes such as containerd. While providing isolation, gVisor adds a significant overhead by making use of an application kernel, which makes it less interesting for edge applications.

**Kata Containers**

Kata Containers is a project that aims to build a secure container runtime by making use of lightweight virtual machines that feel and perform like containers. By doing so, it provides a stronger workload isolation using hardware virtualization technology as a second layer of defense. In contrast to gVisors, Kata Containers are able to achieve a performance that is consistent with standard Linux containers. Kata Containers support industry standards such as OCI Images and the Kubernetes CRI interface, which already makes it compatible with most Kubernetes distributions.

**Firecracker**

Firecracker is a virtual machine monitor that enables easy management of microVMs. Implementing a FLEDGE CRI for Firecracker could open the possibilities to supporting many different kinds of microVMs.

**OSv**

OSv is a unikernel, designed to run unmodified Linux applications securely in the cloud. It is optimized for running on top of a hypervisor and runs everything in a single address space, which makes OSv unikernels very minimal and performant. Besides that, OSv is also capable of assimilating into runtimes such as the Java virtual machine (JVM). This opens the possibility to memory optimizations that are not feasible when running in a different address space than the kernel. Whilst OSv does provide support for JVM Python, Node.JS, Ruby, and Erlang applications out of the box, it is still a general-purpose unikernel.

**Unik**

Unik is a platform for building and running unikernels. It is not a runtime or unikernel by itself, but acts as a toolchain for making working with unikernels easier. It has support for building OSv, Rumprun (Go/C++), Mirage (OCaml), and IncludeOS (C++) unikernels, whilst providing the capabilities of running those unikernels with providers such as Firecracker or QEMU.
Architecture

Building upon FLEDGE, the goal of this master’s dissertation is not only to explore lightweight containerization alternatives, but also to provide a full packaging and deployment solution for these alternatives. This chapter discusses the overall architecture of the solution. In order to not confuse the new solution with FLEDGE, it is named “Feather” (because it is also a lightweight solution).

Firstly, a deep dive is taken into how Feather deploys pods scheduled by the Kubernetes control plane on an edge device in Section 3.1. Secondly, Section 3.2 proposes a general solution for packaging and distributing applications in a runtime-agnostic way. Finally, Section 3.3 gives an overview of how all the components work together.

3.1 Deployment

In Section 2.3 it was mentioned that FLEDGE uses a virtual kubelet provider (the broker) to pass Kubernetes commands to the appropriate Container Runtime Interface. Feather makes use of the same virtual kubelet pattern, but because it is not limited to containers, the CRI used in FLEDGE is referred to as a “backend”. For the same reason, workloads inside a pod are referred to as “instances”.

Both FLEDGE and Feather share a very similar high-level design as shown in Fig. 3.1. The FLEDGE broker is responsible for setting up the node and passing the API server commands to a runtime interface. The runtime interfaces in FLEDGE are responsible for managing the entire workload, from networking to volume management. Feather differentiates itself by incorporating as much pod logic as possible into the provider, which makes it more complicated than the FLEDGE broker. This means that it is not required for a backend to have all the knowledge about the pod itself, since they manage instances independently from each other. As a result, the complexity of Feather backend implementations is reduced.

By using this approach, the provider becomes responsible for singular tasks on the host system, such as networking namespaces, but also for managing the life-cycle of pods, since the backends are only responsible for their instances. A core/v1.PodSpec as passed to the provider contains, besides a set of core/v1.Container’s and core/v1.Volume’s, other information, such as whether host networking is requested on the pod.

The provider extracts the relevant information from the core/v1.PodSpec (e.g. networking, volumes) on a per-instance basis and embeds it into an extended core/v1.Container specification. For example, if the pod is globally configured to
use host networking, the backend still needs that information to create its workloads under that condition. As a result, the backend only needs to receive the extended `core/v1.Container` specification in order to set up the correct configuration for an instance. The backend is then only responsible for pulling images for instances and managing their life-cycles.

### 3.1.1 Virtual kubelet

By making a custom virtual kubelet implementation just like FLEDGE, it becomes possible to decide very meticulously what should happen on every command that is received from the Kubernetes control plane. The virtual kubelet is an adapter that allows us to implement a custom Kubernetes node agent. This virtual kubelet acts as an interposer that calls several functions on the actual provider, which are described in Table 3.1. Since the provider is now responsible for the complete life-cycle of the pods, these functions contain a lot more logic than before. The actual implementation of these functions is further discussed in Chapter 4.

### 3.1.2 Backend

A backend is the component that communicates with the runtime, and as such is responsible for managing the instances themselves. For compatibility purposes, the backend design is inspired by the OCI Runtime Spec, not only because it was designed with individual containers in mind, but because most container runtimes such as containerd tend to adhere to this specification. This specification describes the configuration, execution environment, and life-cycle of a container. The interface for such a backend is described in Table 3.2.

Feather is designed to be generic, such that the specific nature of such a runtime is not relevant. This means that Feather is able to support anything from container runtimes to hypervisors. In order to function as a sufficient proof-of-concept of this design, Feather supports both containerd and OSv respectively. OSv is a general purpose unikernel that has promising optimizations, which makes it an interesting choice to compare it with containerd as a lightweight container runtime.
### 3 Architecture

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ConfigureNode(node)</td>
<td>Populate the specification of the node with extra information.</td>
</tr>
<tr>
<td>CreatePod(pod)</td>
<td>Creates instances for the container definitions in the backends.</td>
</tr>
<tr>
<td>UpdatePod(pod)</td>
<td>Patches a pod with a modified specification.</td>
</tr>
<tr>
<td>DeletePod(pod)</td>
<td>Deletes any trace of the pod.</td>
</tr>
<tr>
<td>GetPod(namespace, name)</td>
<td>Returns the specification of a single pod.</td>
</tr>
<tr>
<td>GetPods()</td>
<td>Returns all pods managed by this provider.</td>
</tr>
<tr>
<td>GetPodStatus(namespace, name)</td>
<td>Returns the status of a single pod.</td>
</tr>
<tr>
<td>GetContainerLogs(namespace, podName, ctrName)</td>
<td>Stream the containers logs from its backend.</td>
</tr>
<tr>
<td>RunInContainer(namespace, podName, ctrName, cmd)</td>
<td>Run the command in the container by using its backend.</td>
</tr>
<tr>
<td>GetStatsSummary()</td>
<td>Return statistics about the node, including running pods.</td>
</tr>
<tr>
<td>GetMetricsResource()</td>
<td>Return metrics that can be collected by Prometheus.</td>
</tr>
</tbody>
</table>

Table 3.1: The interface of the provider in Feather.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GetInstanceStatus(instance)</td>
<td>Returns the status of an instance.</td>
</tr>
<tr>
<td>CreateInstance(instance)</td>
<td>Creates the instance without starting it.</td>
</tr>
<tr>
<td>StartInstance(instance)</td>
<td>Starts the previously created instance.</td>
</tr>
<tr>
<td>UpdateInstance(instance)</td>
<td>Patch an existing instance.</td>
</tr>
<tr>
<td>KillInstance(instance, signal)</td>
<td>Sends the specified signal to the instance.</td>
</tr>
<tr>
<td>DeleteInstance(instance)</td>
<td>Deletes any trace of the instance.</td>
</tr>
<tr>
<td>GetInstanceLogs(instance)</td>
<td>Returns a stream to the logs of an instance.</td>
</tr>
<tr>
<td>RunInInstance(instance, cmd)</td>
<td>Executes a command inside an instance.</td>
</tr>
</tbody>
</table>

Table 3.2: The interface of a backend in Feather.
3 Architecture

containerd

Since containerd is a native container runtime that supports both the OCI Image Spec and the OCI Runtime Spec, it is not complicated to implement a backend for it. In fact, the containerd backend is mostly required to map Kubernetes standards such as core/v1.Container to OCI standards such as the OCI Runtime Spec, such that containerd can interpret it. Furthermore, the image can also be pulled directly by containerd itself, making the backend even more simple.

OSv

Implementing a backend for OSv is not as straightforward. First of all, instead of with a container runtime, the backend needs to communicate with a hypervisor. Consequently, the backend needs to extract a VM disk image from the OCI Image Spec compliant image. This also implies that the backend itself is responsible for pulling the image, which is not the case for containerd. The backend then needs to determine which parameters are required based on the Kubernetes core/v1.Container specification. A hypervisor such as QEMU tends to adhere to its own configuration parameters and standards, and is not designed with the OCI Runtime Spec in mind. Therefore, the OSv backend is responsible for translating the core/v1.Container specification to something that the hypervisor is able to understand. Besides increased complexity, this can result in compatibility issues (e.g. persistent storage and mounts).

3.2 Packaging

When it comes to providing a practical solution for deploying applications in the edge, packaging and distributing them over the network is an important aspect of this architecture. This section proposes a generic solution towards packaging and storing applications in the cloud in order to distribute them easily towards edge devices.

3.2.1 Concept

Packaging non-container images for backend-agnostic storage is a non-trivial problem. For example, OSv images and other microVMs are packaged as raw disk images without any additional information. In order to distribute these atypical “container” images reliably and correctly, well-established container image standards are used in combination with minimal metadata.

Currently, most containers are distributed according to the OCI Distribution and OCI Image specifications in an OCI compliant container registry. In principle, the OCI Distribution specification is designed generically enough to be leveraged as a distribution mechanism for any type of content. This, of course, sparks the interest if the OCI Distribution specification could be used for the backend-specific image (like a disk image). Whilst it is certainly possible to use OCI Artifacts, it is deemed more interesting to store the backend-specific image as an OCI Image for a number of reasons.

- OCI Images are widely accepted as the go-to standard for container images, making them a plug-and-play solution for many use cases (like storing them inside an OCI compliant registry). This means that they are easy to understand by people who are used to working with OCI Images.
3 Architecture

- **OCI** images are generally used by Kubernetes implementations for distributing container images. Using **OCI** images for backend-specific images makes Feather more conform with modern standards.

- **OCI** images store image and configuration options that are consumed by Kubernetes implementations. This allows us to store backend-specific configuration without tampering too much with current standards.

### 3.2.2 Backend

At a high-level an **OCI** implementation (in this case Feather) would normally download an **OCI** image, then unpack that image into an **OCI** Runtime filesystem bundle \([2]\). Feather implements the same concept; however there must be a way to let Feather know how this **OCI** image was packaged in the first way, in order to select the right backend for it.

An example **OCI** image configuration can be seen in Appendix B. According to the **OCI** Image Configuration “Any extra fields in the Image JSON struct are considered implementation specific and MUST be ignored by any implementations which are unable to interpret them.” \([1]\). Thus, in order to let Feather know which backed this image was designed for, it is possible to add extra fields to this configuration.

- **feather.backend** The backend that this image was designed for. Feather reads this and schedules the instance for execution with this particular backend. In case this field is not present, the image is expected to be a standard **OCI** container image and can be scheduled with the default container runtime.

- **feather.runtime (optional)** The preferred runtime to schedule this instance with. In the case of virtual machines, this refers to the used hypervisor.

An **OCI** image configuration extended with those fields added can be seen in Code Fragment 3.1. With these changes in place, Feather is able to deduct the appropriate backend to manage an image with. Since containerd images are already **OCI** Images, these fields are only required for the OSv backend. Whenever the **feather.backend** field is not present, Feather can safely assume a container image. Of course, the actual structure of the image is still left to the implementation and can differ drastically between backends.

Since Feather is only responsible for pulling these images from an **OCI** Distribution Spec compliant registry, an auxiliary tool called “Flint” is created. Flint is a tool that is capable of converting any non-compliant image into an **OCI** image for any of the Feather backends. Furthermore, Flint is also able to push these **OCI** images to any **OCI** Distribution Spec compliant registry, so that they can be pulled by Feather.

### 3.2.3 containerd

Since containerd images are already **OCI** Image Spec compliant, there is no need to make modifications to the current structure. Containerd images can already be pushed to and pulled from **OCI** Distribution Spec compliant registries by default. Although Flint supports the importing, pushing and pulling of container images for completeness, these images can be easily built and pushed with Docker, without modification.
3 Architecture

```
{
  "created": "2015-10-31T22:22:56.015925234Z",
  "author": "Alyssa P. Hacker <alyspdev@example.com>",
  "architecture": "amd64",
  "os": "linux",
  "feather.backend": "osv",
  "feather.runtime": "qemu",
  "config": {
    "User": "alice",
    "ExposedPorts": {
      "8080/tcp": {}
    },
    "Env": [
      "PATH=/usr/local/sbin:/usr/local/bin:/usr/bin:/sbin:/bin",
      "FOO=oci_is_a",
      "BAR=well_written_spec"
    ],
    "Entrypoint": ["/bin/my-app-binary"],
    "Cmd": ["--foreground", "--config", "/etc/my-app.d/default.cfg"],
    "Volumes": {
      "/var/job-result-data": {},
      "/var/log/my-app-logs": {}
    },
    "WorkingDir": "/home/alice",
    "Labels": {}
  },
  "rootfs": {
    "diff_ids": [
      "sha256:c6f988f4874bb0add23a778f753c65efe992244e148a1d2ec2a8b664fb66b8d1",
      "sha256:5f70bf18a086007016e948b04aed3b82103a36bea41755b6cddaf10ace3c3ef"
    ],
    "type": "layers"
  },
  "history": [
    {
      "created": "2015-10-31T22:22:54.690851953Z",
      "created_by": "/bin/sh -c #(nop) ADD file:a3bc1e842b69636f9df5256c49c5374fb4eef1e281fe3f282c65fb853ee171c5 in",
      "empty_layer": true
    },
    {
      "created": "2015-10-31T22:22:55.613815829Z",
      "created_by": "/bin/sh -c #(nop) CMD ["sh"]",
      "empty_layer": true
    }
  ]
}
```

Code Fragment 3.1: Extensions for Feather in an example [OCI](#) Image Configuration [2].
3.2.4 OSv

OSv images are generally built with the Capstan command-line tool. This tool is capable of building virtual machine images from either a Capstanfile or a capstan package. However, these built packages only consist of a single disk image. This, of course, cannot be natively pushed to any OCI Distribution Spec compliant registry.

Fortunately, several options exist for achieving this goal. First of all, the OCI Distribution Spec supports the storage of any kind of media type. This means that the disk image could be stored with a descriptor for a generic application/octet-stream. However, whilst this would technically be possible, this would not leave any space to specify any predefined instance configuration, like is possible with an OCI Image Config. Fortunately, instead of thinking of the disk image as raw binary data, it could be thought of as a single OCI Image Layer. Whilst it is technically not a layer, embedding the disk image as a layer makes the most sense. This allows for just as much configuration as a standard OCI Image without deviating too much from the original OCI Spec.

3.3 Integration

Bringing everything together, the proposed architecture provides a complete solution for developers to deploy OSv images to an edge device which is located in a Kubernetes cluster.

Whenever an image is built with its dedicated packaging tool, Flint is able to convert this to an OCI Image. This OCI Image can then be pushed to any OCI Distribution Spec compliant registry. Being stored in a container registry, it is possible to deploy this OCI Image on an edge device with Kubernetes as usual. By creating a Kubernetes deployment, Feather inspects the Image Config for the core/v1.Container and decides which backend is required to run this particular image. This backend then takes the required actions in order to create and start an instance of this image. This procedure is visualized in Figure 3.2:

![Figure 3.2: Architectural overview of deploying an application with Feather.](image-url)
4

Implementation

This chapter describes the technical implementation choices to achieve the architectural design presented in Chapter 3. This implementation is written in the Go programming language, because a large amount of used technologies are also written in that language. This includes the virtual kubelet, containerd, Kubernetes and Capstan.

The packaging methods for containerd and OSv are covered first in Section 4.1. Secondly, the provider implementation for the virtual kubelet is covered first in Section 4.2. Finally, both the containerd and OSv backend implementations are discussed in Section 4.3.

4.1 Packaging

The packaging solution for Feather is implemented in a separate tool called Flint. Flint is a command-line interface created specifically for storing Feather images inside an OCI Image Registry.

In order to communicate with the OCI Image Registry, Flint makes use of the regclient library. This library also has support to work with local directories that have an OCI Distribution Spec compliant structure. This allows Flint to compose a registry in a local directory, and push these custom images to an OCI Image Registry.

Flint has support for three commands.

- `flint import <backend> <sourceRef> <targetRef>`: Import a non-compliant image from source into a compliant target image. A source reference is formatted into the backend-specific format, whilst the target reference is formatted according to the OCI Distribution Spec.

- `flint pull <ref>`: Pulls an OCI image from an OCI Image Registry into a local directory.

- `flint push <ref>`: Pushes an imported or pulled OCI image to an OCI Image Registry.

This means that, in order to push a non-compliant image to an OCI Image Registry, the `flint import` and the `flint push` commands needs to be run in order. Flint is currently compatible with two Feather backends, “containerd” and “OSv”.


4 Implementation

4.1 containerd

Containerd images are already **OCI** images by nature, so this structure is copied verbatim. These images can be pushed using containerd without Flint, but it is implemented for completeness. An example of pushing a containerd image to a registry can be seen in Code Fragment 4.1.

```
$ ctr images pull docker.io/library/busybox:latest
$ flint import ctd docker.io/library/busybox:latest
INFO [0000] Imported ctd://docker.io/library/busybox:latest to
$ flint push gitlab.ilabt.imec.be:4567/fledge/flint/busybox:latest
to gitlab.ilabt.imec.be:4567/fledge/flint/busybox:latest
```

**Code Fragment 4.1:** Example of pushing a containerd image to an **OCI** image Registry.

4.1.2 OSv

As covered in Section 3.2, OSv needs to be handled very differently than containerd, which is also the reason Flint exists in the first place. Currently, Flint only supports OSv images that are built with Capstan. Flint searches for the specified source image inside the local Capstan repository (`$HOME/.capstan/repository`). Capstan stores a couple of files in a repository, but the only interesting file is the virtual machine disk that is named `<reference>.<hypervisor>`.

In order to create a compliant **OCI** image for this Capstan package, Flint recreates a complete **OCI** Image Registry in a local directory that contains that specific image. To achieve this, Flint follows the following steps.

1. Flint compresses the disk image into a gzipped tar file to prepare it for the **OCI** Image Registry. This is not required, but makes sure that no storage is wasted in the registry with sparse disk images.

2. A new application/vnd.oci.image.layer.v1.tar+gzip descriptor is created in order to store the disk image as an **OCI** Image Layer. The disk image itself is then stored as a blob that is referred to by this descriptor.

3. A descriptor for the **OCI** Image Config is created with the extra "feather.backend": "osv" and "feather.runtime": "<hypervisor>" fields. The rootfs property refers to the disk image instead of an actual layer.

4. This **OCI** Image Config is wrapped together with the **OCI** Image Layer inside a manifest to create the actual **OCI** Image.

5. An **OCI** Index is created for the **OCI** Image to create a repository that can be referred to with `<targetRef>`.

This local repository can then be pushed to any **OCI** Distribution Spec compliant repository to store any OSv image. An example of pushing an OSv image to a registry can be seen in Code Fragment 4.2.
4 Implementation

$ capstan package compose --pull-missing example-image
$ flint import osv example-image
  → gitlab.ilabt.imec.be:4567/fledge/flint/example-image:latest
INFO[0000] Imported osv://example-image to
  → ocidir:///home/mdc/.fledge/images/gitlab.ilabt.imec.be/fledge/flint/example-image:latest
$ flint push gitlab.ilabt.imec.be:4567/fledge/flint/example-image:latest
INFO[0005] Uploaded
  → ocidir:///home/mdc/.fledge/images/gitlab.ilabt.imec.be/fledge/flint/example-image:latest
  → to gitlab.ilabt.imec.be:4567/fledge/flint/example-image:latest

Code Fragment 4.2: Example of pushing an OSv image to an OCI Image Registry.

4.2 Provider

The code for the provider uses the broker pattern from FLEDGE as a starting point. Both making use of the virtual kubelet, they implement the interface discussed in Section 3.1. In Feather, this provider is named the “backend” provider.

Since this provider is aimed towards supporting many backends at the same time, the user is required to configure the enabled backends in order to minimalize overhead. Furthermore, Feather is also able to handle standard OCI images that are not packaged with Flint. In order to determine the backend for these images, Feather can be configured with a default container backend when the feather.backend field is not present. An example of such a configuration can be seen in Code Fragment 4.3.

```
{
  "default": "containerd",
  "enabled": ["containerd", "osv"]
}
```

Code Fragment 4.3: Example configuration for the Feather provider.

The first thing that the provider does besides configuring the node is initializing the backends. For containerd, this includes establishing a connection with the containerd socket. Once these backends are initialized, the provider is able to receive commands from the virtual kubelet.

4.2.1 Interface

Get Pod

The retrieval of a Pod simply returns the core/v1.PodSpec’s that Feather is currently managing.

Get Pod Status

The status of a Pod is constructed from the status of its instances. The appropriate backends are queried for the status of the pod’s instances. These statuses are then cumulated in order to communicate back to the API server whether the pod is pending, starting, running or stopped.
4 Implementation

Create Pod

For creating a new pod, the provider iterates over all containers in the `core/v1.PodSpec` and executes the following steps.

1. Retrieve the [OCI](https://www.oci.io/) ImageConfig from the [OCI](https://www.oci.io/) Image Registry and query the `feather.backend` field. This field decides which backend is responsible for handling any operations on the created instance.

2. Relevant information is extracted from the `core/v1.PodSpec` and inject into the `core/v1.Container` object. This extended specification is called an `Instance` object by Feather. This includes networking options and volumes.

3. The appropriate backend is requested to create the instance with the options from the `Instance` object.

Once all instances are successfully created, they appropriate backend is requested to start them.

Update Pod

The current implementation of Feather deletes and recreates the pod upon update. Whilst this is not performant, it is sufficient for a proof-of-concept.

Delete Pod

When a pod is scheduled for deletion, Feather requests the appropriate backends to delete the instances referenced by this pod.

4.2.2 Expansion

An important part of the provider is to provide the backend with just enough information about the pod in order to manage the instance. The implementation uses an extended `core/v1.Container` object, called the `Instance` object in order to store the extra information, as seen in Code Fragment 4.4.

```go
// An Instance represents a Container with extensions for a Backend
// It's desirable that the backend only knows as much as it needs to set up a Container

type Instance struct {
    ID        string
    Backend   Backend
    +corev1.Container
    VolumeMounts []InstanceVolumeMount
    HostNetwork bool
}
```

Code Fragment 4.4: Extended Container structure, as consumed by a Feather backend.

Currently, the Feather supports the expansion of two fields. Besides communicating to a backend if the pod requests host networking, it also provides a backend with all the information that is required to setup a volume. To achieve this, Feather embeds the `core/v1.Volume` from the `core/v1.PodSpec` inside the `core/v1.VolumeMount` as can be seen in Code Frag-
This renders a backend with enough information about for example which host directory needs to be mounted into the instance.

```go
// InstanceVolumeMount is a VolumeMount with an assigned InstanceVolume
type InstanceVolumeMount struct {
    corev1.VolumeMount
    Volume InstanceVolume
}
```

Code Fragment 4.5: Extended VolumeMount structure, as consumed by a Feather backend.

Unfortunately, this is not enough for some types of volumes. Kubernetes `core/v1.ConfigMap`'s and `core/v1.Secret`'s are stored within the control plane of the cluster. This means that the provider either needs to prefetch these resources, or the backends are required to have knowledge about communicating with the control plane. Following the principle of Feather, it is logical to prefetch these resources when creating an instance. This means that a `core/v1.Volume` needs to be extended with the `core/v1.ConfigMap` and `core/v1.Secret` objects, as can be seen in Code Fragment 4.6.

```go
// InstanceVolume is a Volume with an expanded VolumeSource
// This makes the life of the backend a lot easier and moves work to the provider
type InstanceVolume struct {
    ID string
    corev1.Volume
    Secret *InstanceSecretVolumeSource
    ConfigMap *InstanceConfigMapVolumeSource
}

type InstanceSecretVolumeSource struct {
    *corev1.SecretVolumeSource
    Object *corev1.Secret
}

type InstanceConfigMapVolumeSource struct {
    *corev1.ConfigMapVolumeSource
    Object *corev1.ConfigMap
}
```

Code Fragment 4.6: Extended VolumeMount structure, as consumed by a Feather backend.

### 4.3 Backend

A backend implements the interface described in Section 3.1 and receives requests by the provider. These requests are all provided with an `Instance` object as shown in Code Fragment 4.4. Because the provider is able to handle most of the shared logic between the backends, backends for Feather tends to be very specific. The actual implementation of how a backend handles these requests is thus very different.
4 Implementation

4.3.1 containerd

The containerd backend implementation in Feather is very similar to the implementation present in FLEDGE, except for the parts that are moved to the provider.

Get Instance Status

Since containerd adheres to the OCI Runtime Spec, it is possible to query the status of an instance directly from containerd. This means that the backend is not required to keep track of any of its instances, since containerd stores all required status information internally.

Create Instance

Before requesting the container runtime to create the container, the backend requests containerd to pull the image from the OCI Image Registry. The core/v1.Container object is converted to OCI Runtime Spec options that are understood by the containerd runtime. Containerd is then requested to create a new container with these options. Finally, a task is created for the main process that should be run inside this container. Unfortunately, containerd tasks have no built-in way of storing the logs of the container. To allow the API server to request the logs of the container, the task is requested to write any of its output to a dedicated file for this container.

Start Instance

The task that was previously created for this instance is started by the containerd runtime.

Kill Instance

The task that was previously created for this instance is killed by the containerd runtime with the specified signal.

Delete Instance

Any traces of the container, including its tasks, are deleted from the containerd runtime.

Get Instance Logs

Since containerd does not provide any persistent logging facilities, it is required to use store the output of the task manually inside a file. Upon request by the API server, this log file is then streamed to the client.

Run In Instance

In order to execute a command in the container created by the containerd runtime, a new process is executed inside of the existing task. The output of this process is then directly streamed to the client.
4 Implementation

4.3.2 OSv

Although the OSv backend is used to execute virtual machine images, it is not a lot more complex than the containerd backend due to the design of Feather. Besides being able to package OSv images, Capstan also provides a command-line interface for executing these OSv images. Making use of the library provided by Capstan, it is also possible to minimize the complexity of mapping the instance to hypervisor options.

However, Capstan does not keep any track of when the instance was started or has stopped. In contrast to the containerd backend, the OSv backend is therefore required to keep track of the status of an instance.

Get Instance Status

This communicates the stored status of the instance back to the API server.

Create Instance

Capstan has no built-in way to pull OCI images like containerd has, because an OCI Image structure for OSv images is only specific to Feather. The OSv backend is therefore required to manually pull the single OCI Image Layer from the OCI Image into the local Capstan repository, which is essentially to reverse the operation of Flint. After this operation is complete, the OSv backend creates a pre-configured Capstan instance. This configuration is specific to Capstan and contains the fields as described in Table 4.1.

However, Capstan does not have any native support for volume mounts like containerd has. For mounting any kind of volume inside an OSv instance, the options are quite limited. First of all, mounts are performed under the form of mounting block devices onto a directory. Hereby, OSv has only support for the NFS, ROFS, virtio-fs, and ZFS filesystems. Secondly, because of this behavior, it is only able to mount directories inside the instance. Mounting files directly is practically impossible without any modification.

For the host directory and NFS core/v1.Volume types, virtio-fs and the NFS filesystem can be used respectively. Unfortunately, the current implementation of OSv has only support for read-only virtiofs mounts. This means that the virtiofs filesystem is the only option for persistent storage. Mounting core/v1.Secret’s are less evident, but it is hereby also possible to populate a host directory with all the secrets of an instance and mount it with virtio-fs onto /var/kubernetes/secrets.

However, for mounting single files from e.g. a core/v1.ConfigMap, the solution becomes less evident. In order not to tamper with the OSv source code, it has been decided that it is most ideal to inject a single binary into the OSv image at build time in order to extend the image with these features. This binary, named ‘kubernetes.so’ is responsible for configuring the OSv instance upon initialization by the OSv loader in order to support the Kuberetes features provided by Feather. The current implementation of this binary enables the support for core/v1.ConfigMap’s by copying the directory structure present in /var/run/configmaps onto /. This allows the OSv backend to populate a directory with files mapped by the core/v1.ConfigMap’s and mount it onto /var/run/configmaps in order to provide support for read-only file mounts.

Implementing support for both of these adds entry-level support for read-only directory and file mounts, allowing the
<table>
<thead>
<tr>
<th>Key</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>A unique identifier for the instance.</td>
</tr>
<tr>
<td>Verbose</td>
<td>Whether verbose logging is enabled. Always set to 'true' in Feather.</td>
</tr>
<tr>
<td>Cmd</td>
<td>The command-line that is passed to the OSv image.</td>
</tr>
<tr>
<td>DisableKvm</td>
<td>Whether to not use [KVM] Always set to 'false' in Feather.</td>
</tr>
<tr>
<td>InstanceDir</td>
<td>The directory where the files for this instance are located.</td>
</tr>
<tr>
<td>Monitor</td>
<td>The monitor socket that allows communication to the OSv instance.</td>
</tr>
<tr>
<td>ConfigFile</td>
<td>The file where this config is stored.</td>
</tr>
<tr>
<td>AioType</td>
<td>Which type of asynchronous [IO] to use.</td>
</tr>
<tr>
<td>Image</td>
<td>Which Capstan image to use in order to initialize the instance.</td>
</tr>
<tr>
<td>BackingFile</td>
<td>Whether to use a backing file for this instance disk image. Always set to 'true' in Feather.</td>
</tr>
<tr>
<td>Volumes</td>
<td>Which disks to mount inside the virtual machine. This is not used by Feather because it handles volumes differently.</td>
</tr>
<tr>
<td>Memory</td>
<td>How much memory that is allocated to the virtual machine. Can be limited with resource limits.</td>
</tr>
<tr>
<td>Cpus</td>
<td>How many virtual cpus that are allocated to the virtual machine. Can be limited with resources limits.</td>
</tr>
<tr>
<td>Networking</td>
<td>Networking configuration to configure the virtual machine with.</td>
</tr>
<tr>
<td>Bridge</td>
<td>Which bridge to attach this virtual machine to in case of bridge networking.</td>
</tr>
<tr>
<td>MAC</td>
<td>An optionally fixed [MAC] address for the virtual machine.</td>
</tr>
</tbody>
</table>

Table 4.1: Description of the relevant fields in a Capstan configuration.
configuration of the application that is running inside the instance. These volume mounts are converted to both hypervisor and OSv command-line options, which are appended to the Capstan configuration. The resulting Capstan configuration is then stored inside the instance directory.

Start Instance

The configuration that was created for the instance is read and passed on to Capstan. Capstan makes use of the hypervisor specified in the OCI Image Specification in order to start a virtual machine for this instance. Just like with the containerd backend, the output of this virtual machine needs to be captured in a separate file for future consultation by the client.

Kill Instance

Unlike with the containerd backend, the OSv backend is only able to stop an instance. The specified signal is therefore ignored and the instance is always stopped whenever an instance should be killed.

Delete Instance

In order to delete an instance, the virtual machine is stopped and any traces of its disk image and configuration are removed.

Get Instance Logs

Since Capstan does not provide any persistent logging facilities, it is required to use store the output of the task manually inside a file. Upon request by the API server, this log file is then streamed to the client.

Run In Instance

Due to the limitations of Capstan and the hypervisor, this request is currently ignored by the backend and left unimplemented.
Evaluation

When determining alternative runtimes, the most important aspect for edge computing is reducing resource overhead. However, in the process of using alternatives like microVMs, some features like volume mounts, persistent storage, and networking might not be as straightforward anymore. This Chapter compares the currently implemented containerd and OSv Backends of Feather in order to make a reasonable comparison between containers and microVMs in the aspect of resource utilization.

First of all, Section 5.1 covers the used hardware. Section 5.2 describes the experiment setup and how the results are collected. Thirdly, the results are presented in Section 5.4. Finally, Section 5.5 discusses the implications of using either backend with regard to security, networking and performance.

5.1 Hardware Setup

This evaluation is performed on the imec iLabt Virtual Wall [37]. For the purpose of this experiment, two servers are set up, a master and a worker node. In order to make this set-up as reproducible as possible, Ansible is used to provision all services on both of them.

5.1.1 Master

Specifications: 2x Quad core Intel E5520 (2.2 GHz) CPU, 12 GB RAM, 1x 160 GB harddisk, 2 gigabit nic
Operating system: Ubuntu 20.04.5 LTS (Focal Fossa)

This node takes care of the Kubernetes control plane. Set-up with kubectl, the node runs the primary “node agent”, a default Kubernetes kubelet. The node houses the main components of the control plane [38].

• coredns: General-purpose authoritative DNS server that acts as the cluster DNS server.
• etcd: A distributed, reliable key-value store for the most critical data of the cluster.
• kube-apiserver: Component that exposes the Kubernetes API, which the worker node communicates with.
• kube-controller-manager: Component that runs controller processes.
5 Evaluation

- **kube-scheduler**: Control plane component that watches for newly created Pods with no assigned node, and selects a node for them to run on.

The master node also runs every component of the evaluation setup that is not Feather, such as monitoring systems and visualization, which are discussed in Section 5.2.

5.1.2 Worker

**Specifications**: 2x Quad core Intel E5520 (2.2 GHz) CPU, 12 GB RAM, 1x 160 GB harddisk, 2 gigabit nic
Operating system: Ubuntu 20.04.5 LTS (Focal Fossa)

The second node represents the edge device in a real-world scenario. Only components relevant to the experiment are run on this server.

- **feather 1.0.0**: The Feather service which communicates with the kube-apiserver and is responsible for scheduling containers with their respective backends.

- **containerd 1.6.18**: The container manager needed for the containerd backend. This container manager is the default used by Feather in order to manage standard containers.

- **runc 1.1.4**: This low-level container runtime is used by containerd in order to run the containers.

- **qemu-system-x86 4.2.1**: The hypervisor required for running OSv microVMS.

- **virtiofsd 1.6.0**: The daemon needed for exposing local directories to microVMS by means of virtio-fs.

Besides these components, the node also runs some metrics collectors that are required to monitor the relevant components, which are discussed in Section 5.2.

5.2 Experiment

In order to assess the performance and capabilities of the two Feather backends ‘containerd’ and ‘OSv’, three different scenarios are analyzed.

1. Baseline. Without any instances running, does this backend make use of any resources?

2. Minimal. Does the backed impose a significant overhead when the instance is doing idling?

3. Application benchmark. How well does the backend perform when it is used for a practical application?

Whilst these three scenarios are technically run on a server with 8 physical cores and 12 GB of memory, Feather is designed for edge devices. In order to emulate a realistic edge device with 1 CPU and 1 GiB of RAM, Feather limits containerd and OSv with cgroups and QEMU options respectively.
5 Evaluation

5.2 Baseline

In order to measure the overhead of Feather and the services required by the backends, a scenario without any instances running is analyzed. This means that Feather is doing the minimal amount of work in order to keep everything in sync, and that no pods or containers are scheduled. In this scenario, it is also interesting to discuss the storage footprint in order to compare it with other solutions. With this baseline scenario present, it then becomes possible to analyze how well the solution scales towards running workloads.

5.2.2 Minimal

The scenario for analyzing a minimal workload makes use of a simple Hello World application written in C, which sleeps for the duration of the experiment. This ensures that there is no work done, but that it is still possible to verify if the application has started successfully. The exact code that is used for this experiment can be seen in Code Fragment 5.1.

```c
#include <limits.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    // Disable buffering
    setbuf(stdout, NULL);
    // Say something
    printf("Hello Edge\n");
    // Do nothing, forever
    sleep(INT_MAX);
    return 0;
}
```

Code Fragment 5.1: Source code of the minimal application written in C.

This minimal application is deployed onto the cluster with the Kubernetes command-line `kubectl` for the experiment. The Kubernetes deployment used during the experiment can be found in Code Fragment A.4.

However, the minimal application still needs to be packaged differently for both backends. In order to differentiate between them, two different versions of the application are published to the same repository, as seen in Code Fragment A.4. To keep things simple, Docker [39] and Capstan [40] are used to package images for the containerd and OSv backends respectively. These images are then imported into an OCI compliant registry with `flint` so that they can be deployed on the cluster.

Docker

In order to have a fair advantage towards the binary size of OSv, the Docker image is stripped down to a minimum. As can be seen in Code Fragment A.1 the image is made from scratch, where only the required dynamic libraries are copied over from the Ubuntu image. While it is technically a better choice to compile the application statically in this scenario, many applications are linked dynamically to libc. Therefore, choosing for dynamic compilation covers a broader scenario. In addition, OSv does not have any support for statically compiled applications, so this makes for a more equivalent comparison.
Capstan

To package the application with Capstan, the mpm-pkg format is chosen because it is the most modern [40]. This means that the package is defined through two files, which can be seen in Code Fragment A.2 and Code Fragment A.3. Normally, a minimal application would be considered a package without dependencies. However, the osv.httpserver-monitoring-api package should be considered mandatory since it is the only reliable way to extract relevant information from the microVM. This is a service that is generally run side-by-side with the application on OSv unikernels and enables debugging the command-line, the filesystem, and more. Therefore it is useful to consider this component as an overhead of a minimal OSv image, rather than an overhead of a full application. Note that, although this monitoring API does not add a significant overhead, production environments can omit it to make the image even smaller.

5.2.3 Application

In order to analyze and verify the features of Feather, it is necessary to determine how well Feather and its backends perform under the load of a resource-intensive application. There are several applications that might serve this purpose, but the deployment of a Minecraft game server [41] has been selected as the ideal real-world scenario for several reasons.

• Running on the Java Virtual Machine, a Minecraft game server sufficiently proves that it is possible to run a large application on the desired backend.

• A Minecraft game server is very resource-intensive. Not only does it generate new parts of the world whenever a player travels to a new area, but it also needs to manage all the entities in the world, like pigs, cows, dropped items, and players. Besides that, multiple players might play on the server at the same time, requiring the server to keep up with all of their simultaneous actions. This creates the perfect environment to test the CPU and memory usage of multiple Feather backends under stress.

• The generated world can be stored on disk. This not only allows the verification of several storage options, but also demonstrates the operation of persistent storage.

• The server properties are highly configurable, meaning that native Kubernetes configuration options like ConfigMaps can be tested and used during the experiment.

• Networking is prevalent in the operation of a Minecraft game server. Therefore, this demonstration is able to verify if the instance is able to communicate with devices outside of the cluster.

Because the default Minecraft game server provided by Mojang is not considered to be very performant, the modified PaperMC [42] version is used to perform the analysis. PaperMC is a high-performance modified version of the Minecraft game server that focuses on performance and stability. Furthermore, the default Java Virtual Machine arguments are not ideal for performance. According to the PaperMC documentation, the recommended JVM flags are the ones that were defined by Daniel Ennis (Aikar) [43]. Amongst these flags, the -Xms and -Xmx flags still need tuning. The targeted edge device only has 1 GiB of free memory, therefore it would seem logical to allow the JVM a maximum of 80 % of this free memory.

This still leaves 200 MiB of memory for the operating system, Feather and the backends, which seems like a realistic real-world scenario where most of the memory is used by the application. Because it would be interesting to see how the memory
5 Evaluation

<table>
<thead>
<tr>
<th>Flag</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>-Xms100M</td>
<td>Set initial Java heap size to 100 MiB.</td>
</tr>
<tr>
<td>-Xmx800M</td>
<td>Set maximum Java heap size to 800 MiB.</td>
</tr>
<tr>
<td>-XX:+UseG1GC</td>
<td>Use the Garbage First (G1) Collector.</td>
</tr>
<tr>
<td>-XX:G1ReservePercent=20</td>
<td>Sets the percentage of reserve memory to keep free so as to reduce the risk of to-space overflows.</td>
</tr>
<tr>
<td>-XX:+AlwaysPreTouch</td>
<td>Gets the memory setup and reserved at process start ensuring it is contiguous, improving the efficiency of it more.</td>
</tr>
<tr>
<td>-XX:+DisableExplicitGC</td>
<td>Disable plugins from trying to do perform garbage collection themselves.</td>
</tr>
<tr>
<td>-XX:+ParallelRefProcEnabled</td>
<td>Optimizes the GC process to use multiple threads for weak reference checking.</td>
</tr>
<tr>
<td>-XX:+PerfDisableSharedMem</td>
<td>Disallow the GC to write to the file system, which can cause major latency if disk IO is high.</td>
</tr>
</tbody>
</table>

Table 5.1: Technical explanation of the recommended JVM flags for a PaperMC server.

allocation of the JVM develops in time, -Xms is not set equal to -Xmx, but to a minimum value of about 10 % of the available memory. The relevant flags used in the experiment are explained in Table 5.1. Other flags that are used involve optimizing the garbage collector for Minecraft's high allocation rate of short-lived objects, such as block positions.

It would be evident to run the latest 1.19.4 Minecraft game server for this scenario. However, several problems arose while trying to run any Java version above Java 11 on the OSv backend, which are discussed in Section 5.5. Even though this limits the maximum server version to 1.16.4, the actual server version is not relevant for the analysis. The analysis is done with server version 1.10.2.

Unlike the minimal application, this application allows for configuration. A Minecraft game server is initialized with a seed, where different seeds lead to different worlds to be generated. By keeping the seed consistent between backends, a deterministic testing environment can be created. Of course, by fixating on a single seed, it is possible that the generated world would not represent the generic workings of the Minecraft game server. Therefore, a set of three randomly chosen seeds are be tested in order to be able to interpret the results in a generic way. The chosen seeds can be seen in Table 5.2. Amongst the seed, other properties like the default game mode can be configured. Every single property is configured by mapping the server.properties file with a ConfigMap. The resulting ConfigMap used for the purpose of this experiment can be seen in Code Fragment A.8.

Experiment Repetition Seed

<table>
<thead>
<tr>
<th>Experiment</th>
<th>Repetition</th>
<th>Seed</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>1</td>
<td>-11970185</td>
</tr>
<tr>
<td>*</td>
<td>2</td>
<td>13924251</td>
</tr>
<tr>
<td>*</td>
<td>3</td>
<td>84579735</td>
</tr>
</tbody>
</table>

Table 5.2: Minecraft seeds that are used for initializing the worlds.

The application itself is then deployed to the cluster by applying the definition from Code Fragment A.8. As with the minimal
5 Evaluation

```yaml
apiVersion: v1
class: ConfigMap
metadata:
  name: papermc
data:
  banned-players.json: |
  []
  banned-ips.json: |
  []
  whitelist.json: |
  []
  server.properties: |
    level-seed=<level-seed>
    online-mode=false
    difficulty=1
    enable-command-block=true
    gamemode=1
    use-native-transport=false
```

Code Fragment 5.2: ConfigMap definition for configuring the PaperMC application.

application, this Minecraft game server needs to be packaged differently for the container and the OSv backends. Again, both of these images are then imported into an OCI compliant registry with flint so that they can be deployed on the cluster.

**Docker**

This Docker image is also stripped down to a minimum. Whilst Java requires some linked libraries, it is still able to only provide those, together with the Java binary. Again, in order to keep things fair, the exact same Java and PaperMC builds are used. For the case of the experiment, this is 'Azul Zulu OpenJDK 11.0.19' and 'PaperMC 1.10.2 Build #918'. The resulting Dockerfile is shown in Code Fragment A.5.

**Capstan**

This image is also packaged with the mpm-pkg format provided by Capstan. The difference with the minimal image is that there is now an added Java dependency. Fortunately, OSv has good support for Java applications resulting in an already existing osv.openjdk11-zulu package. However, the produced java binary is not able to follow symlinks towards the system certificates, probably due to filesystem limitations. This is resolved by overwriting javax.net.ssl.trustStore in the boot command. The package and run files are shown in Code Fragment A.6 and Code Fragment A.7 respectively.

### 5.3 Metrics

The Linux kernel exposes many metrics that are useful for these experiments. Amongst those, CPU, memory and disk usage are the most interesting. Whilst these can be monitored through tools like `top` or `htop`, those are not able to produce any meaningful graphs for further analysis. It is therefore necessary to extract these metrics in a useful way so that these can be tracked in time.

Fortunately, the Linux kernel exposes most metrics through the `/proc` filesystem. This filesystem is a pseudoFS designed...
for providing information on the running Linux system. Being a pseudofs, this means that the filesystem does not have actual files. These ‘files’ are virtual entries that provide us a subsystem to the underlying system information.

There are a couple of entries that are relevant. The kernel data files give information about the running kernel. Amongst these, the CPU and memory entries are the most interesting.

- **/proc/cpuinfo**: Information about the CPU.
- **/proc/diskstats**: Disk statistics. This file contains the total bytes read and written from every disk. For the purpose of the analysis, the total number of bytes read and bytes written are taken into account.
- **/proc/meminfo**: Memory info. Amongst other things, contains the available memory, free memory and total memory at the time of reading.
- **/proc/stat**: Overall statistics. Contains lines that represent the total time spent with each CPU core. These fields are formatted as ‘cpuN user nice system idle iowait irq softirq steal guest’. For the actual analysis, the total time a CPU is used is calculated as the sum of every field except the time spent idle.

Furthermore, it would be desirable to collect per-process statistics. The /proc filesystem exposes these statistics through their respective /proc/<pid> directory. Each one of these process directories contains a relevant entry.

- **/proc/<pid>/stat**: Process status. Contains user time, system time, resident memory and virtual memory usage, which is everything that is required for the analysis.

Before going further into these three experiments, it is important to distinguish the different metrics, components and processes that are measured or are accounted as overhead for these backends.

### containerd

For containerd, only runc and containerd itself are monitored as overhead. However, since the application is technically running on the host operating system itself, it is important that the processes inside the container are also monitored and accounted for.

- **/usr/bin/containerd**: The daemon of the container manager.
- **/usr/bin/containerd-shim-runc-v2**: The intermediary between the container manager (containerd) and the actual container runtime (runc).

### OSv

OSv is more difficult to monitor. First of all, a hypervisor is started by Feather using the microVM image. In this case qemu is used to start the microVM. Since the application is running as a microVM, the processes are not actually visible on the host operating system, but it is sufficient to monitor just qemu for the profiling of the whole instance. Furthermore, virtiofsd is required in order to provide bind mounts inside the microVM. Therefore, both are monitored in order to get a realistic profiling of the OSv backend.
5 Evaluation

- /usr/bin/qemu-system-x86_64: The used system virtualizer.
- /usr/bin/virtiofsd: A virtio-fs vhost-user device daemon, used for exposing directories as a virtio-fs inside the microVM.

It must be noted that, whilst it is not directly possible to monitor the processes inside OSv, OSv also ships with a monitoring API that exposes some metrics about OSv itself. Because this evaluation is mostly interested in the memory overhead on the host-end, this monitoring API is not useful in this context, but it is important to note that it exists.

5.3.1 Prometheus

In order to collect and process all these metrics, they are not extracted manually from the /proc filesystem directly. Instead, Prometheus is used. Prometheus is a popular open-source monitoring toolkit. Prometheus collects and stores its metrics as time series data and has support for flexible queries in PromQL. Besides the default metrics stored by Prometheus, there are a numerous amount of libraries and servers which help exporting existing metrics as Prometheus metrics. These are called exporters. By using exporters, it is possible to extract all the relevant information from the /proc filesystem and more.

Node Exporter

The Prometheus Node Exporter is an official exporter for hardware and OS metrics exposed by the Linux kernel. This exporter provides statistics for analyzing the CPU usage and the memory usage. It has support for various collectors, but the following are most relevant, which exports their information from the /proc filesystem as discussed in Subsection 5.3.

- cpu: Exposes CPU statistics.
- diskstats: Exposes disk I/O statistics.
- meminfo: Exposes memory statistics.

Whilst the Prometheus Node Exporter usually runs inside a container managed by Kubernetes, the exporter runs on the host itself in order to not tamper with the measurements of the containerd backend.

Process Exporter

The Prometheus process exporter collects per-process statistics from the /proc filesystem. These statistics contain information about CPU, memory and disk usage, as mentioned in Subsection 5.3. This exporter is used to monitor the resource usage of the following processes on the worker node.

- containerd
- dockerd
- fledge
- libvirtd
5 Evaluation

• process-exporter
• prometheus-node
• qemu-system-x86
• virtiofsd

Golang Exporter [48]

In order to collect more specific statistics about Feather than CPU, memory and disk usage, the Feather application can be instrumented with the Golang exporter. This exporter exposes metrics from the application such as the amount of goroutines, a breakdown of the memory usage, and information about the garbage collector.

This information can either be used as a verification of the process metrics, or for further exploration of the memory usage of the Feather application.

Minecraft Exporter [49]

Because the application scenario makes use of the JVM, it is interesting to explore the memory statistics of the JVM itself. Although the maximum memory of the JVM is fixed at 800 MiB, the JVM is still able to decide how much memory to allocate for the Minecraft game server. In addition, it is useful to monitor the actual memory that is occupied by the Minecraft game server regardless of how much memory is allocated. Fortunately, there is a PaperMC server plugin that, among other things, exports these Java Virtual Machine metrics. Therefore, it is useful to include this plugin in the application scenario images. This not only adds the ability to monitor the memory allocation of the JVM in time, but also adds the ability to measure some Minecraft game server statistics such as the responsiveness.

OSv Monitoring API

OSv provides a useful API for debugging applications. However, for the purpose of this experiment, it is not used to collect any metrics since the Minecraft Exporter provides enough information for evaluation.

5.4 Results

In order to have a fair comparison between the three scenarios, the system is rebooted at the start of every scenario. Besides the scenarios themselves, no non-system processes are left running on the worker node. For every single scenario, the resource usage in terms of CPU and memory are discussed. In case of the baseline scenario, the storage requirements for the backends are also discussed.
5 Evaluation

5.4.1 Baseline

Feather

The Feather binary used for the experiment is a stripped, statically linked binary that is built for the x86_64 architecture of the worker node. This binary is 53.56 MiB in size, around 15 MiB larger than the FLEDGE binary, which is 37.86 MiB. In comparison, a single KubeEdge v1.13.0 agent is 68.99 MiB in size.

In order to investigate the main cause of this increase in size, it is possible to analyze which libraries contribute the most. The first thing that stands out in Table 5.3 is the increase in spaces used by Kubernetes packages. Whilst it is not clear where this added size specifically comes from, it is assumed that this is caused by Feather having seemingly unimportant imports that include extra dependencies such as Kubernetes APIs not part of FLEDGE. Feather also includes regclient and capstan, which in turn have their own dependencies that contribute to the binary size. Since there are no further major contributions besides Kubernetes, it can be concluded that the increase in binary size is caused by a couple of factors, including the additional OSv backend.

<table>
<thead>
<tr>
<th>Package</th>
<th>Size in FLEDGE</th>
<th>Size in Feather</th>
</tr>
</thead>
<tbody>
<tr>
<td>*</td>
<td>19 MiB</td>
<td>26 MiB</td>
</tr>
<tr>
<td>k8s.io/*</td>
<td>9.7 MiB</td>
<td>12 MiB</td>
</tr>
<tr>
<td>k8s.io/api/*</td>
<td>6.4 MiB</td>
<td>7.1 MiB</td>
</tr>
<tr>
<td>k8s.io/api/core/v1</td>
<td>1.8 MiB</td>
<td>1.9 MiB</td>
</tr>
<tr>
<td>k8s.io/client-go/*</td>
<td>2.4 MiB</td>
<td>2.7 MiB</td>
</tr>
<tr>
<td>k8s.io/client-go/applyconfigurations/*</td>
<td>0.94 MiB</td>
<td>1.0 MiB</td>
</tr>
<tr>
<td>github.com/*</td>
<td>3.5 MiB</td>
<td>6.1 MiB</td>
</tr>
<tr>
<td>github.com/google/*</td>
<td>-</td>
<td>2.1 MiB</td>
</tr>
<tr>
<td>github.com/containerd/*</td>
<td>1.4 MiB</td>
<td>0.83 MiB</td>
</tr>
<tr>
<td>google.golang.org/*</td>
<td>1.1 MiB</td>
<td>1.4 MiB</td>
</tr>
<tr>
<td>go/*</td>
<td>-</td>
<td>1.2 MiB</td>
</tr>
<tr>
<td>go/self/*</td>
<td>-</td>
<td>1.2 MiB</td>
</tr>
</tbody>
</table>

Table 5.3: Packages with a significant contribution to FLEDGE’s and Feather’s binary sizes (>1.0 MiB).

The only relevant processes consuming resources on the worker node are feather and containerd. Figure 5.3 shows that the CPU usage of feather has an initial peak at around 23%, dropping to an average of 1.7%. Feather uses an average of 98 MiB of memory, significantly more than FLEDGE, which is able to achieve an average of 50 MiB of memory while idling on x86_64 architectures 37.

ccontianerd

The binaries and dependencies for containerd require 116 MiB of storage, as reported by the APT package manager on a minimal Ubuntu 20.04.05 LTS installation.
5 Evaluation

As seen in Figure 5.1, containerd has no significant CPU usage while idle, and an average of 55 MiB of memory.

OSv

The binaries and dependencies for QEMU require 371 MiB of storage, as reported by the APT package manager on a minimal Ubuntu 20.04.05 LTS installation. Furthermore, virtiofsd requires 5.3 MiB of storage, resulting in a cumulative storage requirement of 376 MiB for the OSv runtime dependencies. A pre-built QEMU hypervisor package depends on many components that provide a wide range of features, such as UI drivers. QEMU can be built from source to reduce this storage overhead, excluding all features that are non-essential for the OSv backend.

However, neither qemu-system-x86_64 or virtiofsd are running if there is no managed instance present, so in this case the OSv backend is not consuming any resources.

5.4.2 Minimal

In the case of the minimal example, it is not relevant to look at the resource usage by the example application itself, since the main application is just idling. However, it is interesting to verify how well the resource usage scales in comparison with the baseline scenario.

Feather

It is important to note that the minimal scenario is run once for every backend, because it is interesting to analyze the resource usage by Feather in both cases. In both cases, the container is created at 0:30. Figure 5.2 shows that the containerd uses some extra CPU at container creation, while OSv does not. However, it is possible that is is because the containerd backend involves complex communication with an external daemon, whilst the OSv backend only needs to prepare the configuration and disk files for the instance. In both scenarios, there is only some additional memory usage during pod creation.
5 Evaluation

at 0:30.

**containerd**

Creating the minimal Docker image for results in an OCI Image of 1.07 MiB. Because the minimal binary that is included is only in 16 KiB size, this is still fairly large.

As seen in Figure 5.2, it is notable that both containerd and containerd-shim-runc-v2 use an insignificant amount of processing resources, except at container creation, when containerd uses 4.2% of the CPU. containerd uses 56.45 MiB of memory on average, while containerd-shim-runc-v2 uses around 9.46 MiB. This results in a total memory overhead of 65.76 MiB on average.

**OSv**

Packaging the minimal Capstan image with Flint results in an OCI Image of 4.23 MiB. This is a 295% increase over the image created for the containerd backend, which is significant. The OSv image also includes the unikernel libraries and the monitoring API, so this result is not unexpected.

The QEMU hypervisor directly uses a significant amount of processing power, idling at 11.5%. With a memory consumption of 63.28 MiB on average, QEMU does perform slightly better than containerd and containerd-shim-runc-v2 combined.

Note that there are no metrics for virtiofsd included in Figure 5.2. This is because Feather did not start virtiofsd, as the minimal example does not make use of any volume mounts.

![Figure 5.2: CPU and memory usage of relevant Feather processes in the minimal scenario.](image-url)
5 Evaluation

5.4.3 Application

The application scenario is performed six times. It is done three times for each level-seed specified in Table 5.2, and this for every backend.

To ensure representative results, the tests are performed as follows.

1. Any previous deployment is deleted.
2. The server image is deployed on the worker node using Kubernetes. The logs then state the startup time of this server, and verify the configuration of the level-seed through the core/v1.ConfigMap.
3. After 2 minutes, 4 clients join the server at the same time. This verifies the ability of the server to handle simultaneous events, such as chunk generation which stresses the CPU.
4. After 4 minutes, villager entities are spawned at a rate of about 3 villagers per 10 in-game ticks. On a server that is fully responsive at 20 in-game ticks per second, this corresponds to a rate of 6 villagers per second.

The course of the tests is shown in Figure 5.3. It is clear that the responsiveness of the server (expressed in ticks per second or TPS) drops slightly after 2 minutes, and drops steadily after 4 minutes. The rate at which villagers are spawned also decreases in time, which notes its relation to the TPS as mentioned in Step 4.

Creating the Docker image results in an OCI Image of 66.03 MiB, while the Capstan image is only 60.23 MiB in size. Since both images are compressed with gzip, a decrease of 8.8 % must be the result of the layered structure of the Docker image, which is not the case for the Capstan image that has only a single “layer”.

As interpreted from the logs of the instances, the contained instance starts in 38.713s, while the OSv instance takes 37.199s to start, which is 18% faster. In the second and third tests, the OSv instance has an startup time with respective improvements of 8 % and 18 % over the containerd instance.

It is however noticeable in Figure 5.3 that the TPS drop slightly faster for the OSv backend than for the containerd backend, which could signify that the OSv backend is slightly slower. However, as can be seen in Figure 5.4, it is clear that qemu-system-x86_64 does not adhere to the 1 virtual CPU limit that is enforced in the Kubernetes deployment. Figure 5.4 shows that qemu-system-x86_64 is using even more than 150 % of CPU usage in the second test. Even though qemu-system-x86_64 does use about 30 %-50 % more CPU than the java process running inside the container, the TPS shows this does not result in better performance.

OSv is consuming less resources at 8:30 than contained at 7:35, where they are both managing an equal amount of 1200 villagers in the first test. As shown in Fig. 5.6, OSv is only consuming 155 MiB of the allocated 224 MiB, while contained already uses about 188 MiB of the allocated 288 MiB of memory a minute earlier. Unfortunately, OSv performs worse with a TPS of 7.84, in comparison to contained with a TPS of 11.23, as shown in Fig. 5.3.

By the end of the first test, qemu-system-x86_64 uses only 661.10 MiB of RAM, which is remarkably less than its container counter-part, which is using 820.57 MiB of memory at that time. Even greater improvements can be seen in the two subsequent tests in Figure 5.4. When looking at JVM statistics, Figure 5.5 shows that the OSv image spends a lot less time on
garbage collection, and that it uses a minimal amount of allocated memory over the course of all tests. The containerd instance however, allocates significantly more memory and has a more jagged “used memory” curve. This result is expected, since OSv claims that it is able to expose extra hardware features to the JVM because it is running in the same address space. This allows the JVM to manage memory much more efficiently, resulting in less allocations and time spent on garbage collection, as is shown in Figure 5.3 and Figure 5.6.

Figure 5.3: Ticks per second in comparison with the number of villagers in the application scenario.

Figure 5.4: CPU and memory usage of relevant Feather processes in the application scenario.
5 Evaluation

5.5 Analysis

5.5.1 Non-functional Requirements

This evaluation is performed for a reference edge device that has 1 CPU and 1GiB of memory as a reference.

Evidently, QEMU does seem to use more CPU than the CPU limits that are set by Kubernetes. QEMU also introduces some CPU overhead that is not attributed to the application as seen in the minimal scenario. This means that Feather is required to implement cgroup management for the OSv backend in order to have a true limitation of the CPU resources that QEMU is able to consume. An alternative solution is to use a different hypervisor than QEMU which does not have this issue. In a real scenario, QEMU would of course not be able to consume more processing power than is present. This might even result in slightly worse performance with regard to TPS.

In terms of memory, QEMU far outperforms the containerized Java application. The OSv instance requires 20% less memory than the containerd instance. This improved memory management is also clearly visible in the increased “used memory”
stability of the JVM. Because of the hard limit on memory resources, the OSv image is certainly a better choice for Java applications that are memory intensive.

When making the choice to support OSv instances on the worker node, the storage requirements are also very similar to these of containerd. It must be noted that supporting both backends on a single node does require about double the storage.

5.5.2 Functional Requirements

It is shown that Feather enables the configuration and deployment of both containerd and OSv images with Kubernetes. Since Kubernetes already supports containerd natively, it is evident that the containerd backend has the potency to support everything that Kubernetes has to offer. However, not everything is compatible with the OSv backend, due to hypervisor or OSv limitations.

Compatibility

While OSv claims to be a Linux binary compatible unikernel, it is not proven that this statement is entirely true. It was not possible to construct OSv images with a version greater than 11, contrary to what is claimed by the developers of OSv. Furthermore, OSv does not contain binaries that are evident in GNU/Linux operating systems such as /bin/sh, which can result in an exception as seen in Code Fragment 5.3. Certain symbols also seem to be missing in shared libraries provided in the OSv unikernel, such as posix_spawn, as seen in Code Fragment 5.4. Despite this, OSv does provide great support for Java, Python, Node.js, Ruby and Erlang applications in general, as is proven by the results of this evaluation.

```
Loading libraries, please wait...
[ERROR] Failed to construct terminal; falling back to unsupported java.io.IOException: Cannot run program "sh": error=2, No such file or directory
```

Code Fragment 5.3: Missing shell executable reported by the OSv instance in the application scenario (seed 1).

```
... 
/usr/lib/jvm/jdk-zulu11.64.19-ca-jdk11.0.19-linux_x64/lib/server/libjvm.so: ignoring missing symbol fma
/usr/lib/jvm/jdk-zulu11.64.19-ca-jdk11.0.19-linux_x64/lib/server/libjvm.so: ignoring missing symbol setutxent
/usr/lib/jvm/jdk-zulu11.64.19-ca-jdk11.0.19-linux_x64/lib/server/libjvm.so: ignoring missing symbol getutxent
/usr/lib/jvm/jdk-zulu11.64.19-ca-jdk11.0.19-linux_x64/lib/server/libjvm.so: ignoring missing symbol malloc_trim
/usr/lib/jvm/jdk-zulu11.64.19-ca-jdk11.0.19-linux_x64/lib/server/libjvm.so: ignoring missing symbol waitid
/usr/lib/jvm/jdk-zulu11.64.19-ca-jdk11.0.19-linux_x64/lib/libjava.so: ignoring missing symbol posix_spawn
... 
```

Code Fragment 5.4: Missing symbols reported by the OSv instance in the application scenario (seed 1).
5 Evaluation

Configuration

Configuration through ConfigMaps and Secrets is greatly supported by the OSv backend, but the only requirement is that the OSv image does include the custom kubernetes.so extension that enables this functionality.

Storage

In Kubernetes environments, it is a common practice to create core/v1.PersistentVolume objects in order to store data persistently. However, Feather has no support for core/v1.PersistentVolume's, which is a limitation of the Virtual Kubelet itself.

It is mentioned before that QEMU is only able to create bind mounts for directories through virtio-fs, but not for files. In addition, the virtio-fs driver provided by the OSv kernel only supports read-only mounts. This means that it is impossible to use any bind mount for persistent storage in the OSv backend. In order to create a read-only bind mount for files, the custom kubernetes.so library also needs to be included in the image. The only filesystems that also have drivers, and can be used for persistent storage, are NFS and ZFS. Since Kubernetes only supports NFS filesystems as a core/v1.VolumeSource, this seems like the only option to store files persistently for the OSv backend.
Future Work

Besides being able to provide most of the Kubernetes features for both containerd and OSv images, Feather is still far from a complete solution. Being a proof-of-concept, Feather proves that it is possible to extend the current FLEDGE implementation beyond just containers. Built with the goal of making backends as simple as possible, Feather enables the efficient implementation of new backends, or new functionality. At this moment, the OSv backend has some functionality limitations that might render it inadequate for certain use cases, such as using writable bind mounts. Support for hypervisors other than QEMU can also be provided to overcome its current issues and limitations. Other than OSv, new backend implementations for e.g. WASM, Kata containers, Firecracker, or Unikraft could further extend Feather with a new range of possibilities. It could be possible that these new backends have some limitations of their own, but this allows the developer to choose the most adequate backend with regard to both non-functional and functional requirements.

Unfortunately, this master’s dissertation does not focus on any networking features or capabilities in the edge. Feather only supports minimal internet access and allows the developer to configure NAT for the application, in order to reach it from outside the Kubernetes cluster. This means that, besides implementing more networking functionality such as bridge networking, services and proxies, there are still many options to explore for optimization.

Even though Feather is not complete, it is already possible to incorporate this solution in production environments. The existing features are already sufficient for deploying OSv images in simple environments that do not require persistent storage or networking functionality besides NAT. It must be noted that Feather should be tested more thoroughly before eventually incorporating it, because Chapter 5 describes the only reasonable testing that has been done so far.
Conclusion

Feather provides a solution for extending the Kubernetes ecosystem with non-container alternatives such as unikernels, which can result in a reduced resource overhead and more secure virtualization. By leveraging existing standards such as the OCI Specifications and the Kubernetes ecosystem, this master’s dissertation ensures that the proposed solution is seamlessly interoperable with the current infrastructure, requiring minimal effort by the developer to leverage the capabilities of Feather.

At the root of this ease of use lies the creation of Flint, which allows a developer to store any type of custom image inside an OCI Image Registry, such as DockerHub or GitLab Container Registry. By asking Kubernetes to deploy this image onto an edge device, Feather is then capable of performing everything that is necessary to execute this image on this edge device. Besides using Flint to repackage the image to an OCI Image format, the developer does not need to change the standard Kubernetes workflow in order to leverage the capabilities of running unikernels in the edge.

In its current state, Feather can be considered production-ready, besides some limitations in features. Being built on the Kubernetes ecosystem, companies are able to efficiently orchestrate performant unikernels between edge devices. As a result, it is possible to take full advantage of the available edge computing resources. Whenever edge devices are not using all of their resources, instances can be automatically make use of these free resources, further supporting the osmotic computing paradigm.

In summary, this master’s dissertation has made great improvements upon FLEDGE, extending this Kubernetes agent to non-container runtimes, such as OSv. This work opens up possibilities for further research and development, including the research towards more lightweight container alternatives for deploying Kubernetes workloads in the edge.
References


References


[27] K0s | kubernetes distribution for bare-metal, on-prem, edge, iot. [Online]. Available: https://k0sproject.io/


References


Appendices
Appendix A

Deployment and build configs for evaluation.

Minimal

FROM scratch as runner

# Copy required dynamic libraries
COPY --from=ubuntu /
   /lib/x86_64-linux-gnu/libc.so.6 
   /lib/x86_64-linux-gnu/
COPY --from=ubuntu /
   /lib64/ld-linux-x86-64.so.2 \ 
   /lib64/

# Add minimal executable
COPY main /main

# Start minimal executable
CMD ["/main"]

Code Fragment A.1: Dockerfile for the minimal application.

name: minimal
title: Minimal benchmark
author: Maxim De Clercq
created: "2023-04-13T18:29:52+02:00"
require:
  - osv.httpserver-monitoring-api

Code Fragment A.2: Capstan meta/package.yaml for the minimal application.

runtime: native
config_set:
  default:
    bootcmd: /main
config_set_default: default

Code Fragment A.3: Capstan meta/run.yaml for the minimal application.
```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: minimal-{{packager}}
spec:
  selector:
    matchLabels:
      run: minimal-{{packager}}
  replicas: 1
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
  template:
    metadata:
      labels:
        run: minimal-{{packager}}
    spec:
      containers:
      - name: app
        image: gitlab.ilabt.imec.be:4567/fledge/benchmark/minimal:v1.0.0-{{packager}}
        imagePullPolicy: Always
        resources:
          limits:
            cpu: 1
            memory: 1Gi
        nodeSelector:
          type: virtual-kubelet
  tolerations:
  - key: virtual-kubelet.io/provider
    operator: Equal
    value: backend
    effect: NoSchedule
```

Code Fragment A.4: Deployment definition for the minimal application.
Appendices

Application

# Inspired by
FROM ubuntu:latest as builder

# Add Java Runtime Environment 11
ARG JRE11_URL
ADD ${JRE11_URL} /usr/lib/jvm/java-11-openjdk/download.tar.gz
RUN tar --extract --gzip --file=/usr/lib/jvm/java-11-openjdk/download.tar.gz --directory=/usr/lib/jvm/java-11-openjdk --strip-components=1
RUN rm /usr/lib/jvm/java-11-openjdk/download.tar.gz
FROM scratch as runner

# Copy required dynamic libraries
COPY --from=builder /lib/x86_64-linux-gnu/libc.so.6 /lib/x86_64-linux-gnu/libdl.so.2
COPY --from=builder /lib/x86_64-linux-gnu/libgcc_s.so.1 /lib/x86_64-linux-gnu/libm.so.6
COPY --from=builder /lib/x86_64-linux-gnu/libpthread.so.0 /lib/x86_64-linux-gnu/librt.so.1
COPY --from=builder /lib/x86_64-linux-gnu/libz.so.1 /lib/x86_64-linux-gnu/

# Copy Java Development Kit 11
COPY --from=builder /usr/lib/jvm/java-11-openjdk /usr/lib/jvm/java-11-openjdk
ENV PATH=${PATH}:/usr/lib/jvm/java-11-openjdk/bin/
ENTRYPOINT ["/usr/lib/jvm/java-11-openjdk/bin/java"]

# Add Minecraft game server
ARG PAPER_URL
ADD ${PAPER_URL} /paper.jar

# Add plugins
COPY plugins /plugins

# Add default settings
COPY eula.txt ops.json server.properties /

# Start Minecraft server
CMD ["-Xmx100M", "-Xms800M", "-XX:+UseG1GC", "-XX:+ParallelRefProcEnabled",
"-XX:MaxGCPauseMillis=200", "-XX:+UnlockExperimentalVMOptions", "-XX:+DisableExplicitGC",
"-XX:AlwaysPreTouch", "-XX:G1NewSizePercent=30", "-XX:G1NewSizePercent=40",
"-XX:G1HeapRegionSize=8M", "-XX:G1ReservePercent=20", "-XX:G1HeapWastePercent=5",
"-XX:G1MixedGCCountTarget=4", "-XX:InitiatingHeapOccupancyPercent=15",
"-XX:G1MixedGCLiveThresholdPercent=90", "-XX:G1RSetUpdatingPauseTimePercent=5",
"-XX:SurvivorRatio=32", "-XX:PerfDisableSharedMem", "-XX:MaxTenuringThreshold=1",
"-Dusing.aikars.flags=https://mcflags.emc.gs", "-Daikars.new.flags=true", "-jar",
"/paper.jar"]


name: papermc
title: High-performance Minecraft game server
author: Maxim De Clercq
created: "2023-04-13T18:29:52+02:00"
require:
- osv.httpserver-monitoring-api
- osv.openjdk11-zulu

runtime: native
cfg_set:
default:
  bootcmd: >
    /usr/lib/jvm/java/bin/java
    -Djavax.net.ssl.trustStore=/etc/pki/ca-trust/extracted/java/cacerts
    -Xms100M
    -Xmx800M
    -XX:+UseG1GC
    -XX:+ParallelRefProcEnabled
    -XX:MaxGCPauseMillis=200
    -XX:+UnlockExperimentalVMOptions
    -XX:+DisableExplicitGC
    -XX:+AlwaysPreTouch
    -XX:G1NewSizePercent=30
    -XX:G1MaxNewSizePercent=40
    -XX:G1HeapRegionSize=8M
    -XX:G1ReservePercent=20
    -XX:G1HeapWastePercent=5
    -XX:G1MixedGCCountTarget=4
    -XX:InitiatingHeapOccupancyPercent=15
    -XX:G1MixedGCLiveThresholdPercent=90
    -XX:G1RSetUpdatingPauseTimePercent=5
    -XX:SurvivorRatio=32
    -XX:+PerfDisableSharedMem
    -XX:MaxTenuringThreshold=1
    -Dusing.aikars.flags=https://mcflags.emc.gs
    -Daikars.new.flags=true
    -jar /paper.jar

cfg_set_default: default

Code Fragment A.7: Capstan meta/run.yaml for the PaperMC application.
apiVersion: apps/v1
kind: Deployment
metadata:
  name: papermc-{{packager}}
spec:
  selector:
    matchLabels:
      run: papermc-{{packager}}
  replicas: 1
strategy:
  type: RollingUpdate
  rollingUpdate:
    maxSurge: 1
template:
  metadata:
    labels:
      run: papermc-{{packager}}
spec:
  containers:
  - name: app
    image: gitlab.ilabt.imec.be:4567/fledge/benchmark/papermc:1.10.2-{{packager}}
    imagePullPolicy: Always
    resources:
      limits:
        cpu: 1
        memory: 1Gi
    ports:
    - containerPort: 9940
      hostPort: 9940
    - containerPort: 25565
      hostPort: 25565
  volumeMounts:
  - name: config
    mountPath: /
    readOnly: true
  volumes:
  - name: config
    configMap:
      name: papermc
nodeSelector:
  type: virtual-kubelet
tolerations:
  - key: virtual-kubelet.io/provider
    operator: Equal
    value: backend
    effect: NoSchedule

Appendices

Appendix B

Excerpt of the OCI Image Spec [3].

The OpenContainers Image Spec Config

OCI Image Configuration

An OCI Image is an ordered collection of root filesystem changes and the corresponding execution parameters for use within a container runtime. This specification outlines the JSON format describing images for use with a container runtime and execution tool and its relationship to filesystem changesets, described in Layers.

This section defines the application/vnd.oci.image.config.v1+json media type.

Terminology

This specification uses the following terms:

Layer

- Image filesystems are composed of layers.
- Each layer represents a set of filesystem changes in a tar-based layer format, recording files to be added, changed, or deleted relative to its parent layer.
- Layers do not have configuration metadata such as environment variables or default arguments - these are properties of the image as a whole rather than any particular layer.
- Using a layer-based or union filesystem such as AUFS, or by computing the diff from filesystem snapshots, the filesystem changeset can be used to present a series of image layers as if they were one cohesive filesystem.

Image JSON

- Each image has an associated JSON structure which describes some basic information about the image such as date created, author, as well as execution/runtime configuration like its entrypoint, default arguments, networking, and volumes.
- The JSON structure also references a cryptographic hash of each layer used by the image, and provides history information for those layers.
- This JSON is considered to be immutable, because changing it would change the computed ImageID.
- Changing it means creating a new derived image, instead of changing the existing image.

Layer DiffID

A layer DiffID is the digest over the layer’s uncompressed tar archive and serialized in the descriptor digest format, e.g., sha256:a9561eb1b198625c9adb5a9513e72c4dedaf9c1cb2d4c5236c9a6957ec7d7df5a9. Layers SHOULD be packed and unpacked reproducibly to avoid changing the layer DiffID, for example by using tar-split to save the tar headers.
NOTE: Do not confuse DiffIDs with layer digests, often referenced in the manifest, which are digests over compressed or uncompressed content.

Layer ChainID

For convenience, it is sometimes useful to refer to a stack of layers with a single identifier. While a layer's DiffID identifies a single changeset, the ChainID identifies the subsequent application of those changesets. This ensures that we have handles referring to both the layer itself, as well as the result of the application of a series of changesets. Use in combination with rootfs.diff_ids while applying layers to a root filesystem to uniquely and safely identify the result.

Definition

The ChainID of an applied set of layers is defined with the following recursion:

\[
\text{ChainID}(L_0) = \text{DiffID}(L_0)
\]
\[
\text{ChainID}(L_0|\ldots|L_{n-1}|L_n) = \text{Digest}(\text{ChainID}(L_0|\ldots|L_{n-1}) + " " + \text{DiffID}(L_n))
\]

For this, we define the binary | operation to be the result of applying the right operand to the left operand. For example, given base layer A and a changeset B, we refer to the result of applying B to A as A|B.

Above, we define the ChainID for a single layer (L₀) as equivalent to the DiffID for that layer. Otherwise, the ChainID for a set of applied layers (L₀|...|Lₙ₋₁|Lₙ) is defined as the recursion

\[
\text{Digest}(\text{ChainID}(L_0|\ldots|L_{n-1}) + " " + \text{DiffID}(L_n)).
\]

Explanation

Let's say we have layers A, B, C, ordered from bottom to top, where A is the base and C is the top. Defining | as a binary application operator, the root filesystem may be A|B|C. While it is implied that C is only useful when applied to A|B, the identifier C is insufficient to identify this result, as we'd have the equality C = A|B|C, which isn't true.

The main issue is when we have two definitions of C, C = C and C = A|B|C. If this is true (with some handwaving), C = x|C where x = any application. This means that if an attacker can define x, relying on C provides no guarantee that the layers were applied in any order.

The ChainID addresses this problem by being defined as a compound hash. We differentiate the changeset C, from the order-dependent application A|B|C by saying that the resulting rodfs is identified by ChainID(A|B|C), which can be calculated by ImageConfig.rootfs.

Let's expand the definition of ChainID(A|B|C) to explore its internal structure:

\[
\text{ChainID}(A) = \text{DiffID}(A)
\]
\[
\text{ChainID}(A|B) = \text{Digest}(\text{ChainID}(A) + " " + \text{DiffID}(B))
\]
\[
\text{ChainID}(A|B|C) = \text{Digest}(\text{ChainID}(A|B) + " " + \text{DiffID}(C))
\]

We can replace each definition and reduce to a single equality:

\[
\text{ChainID}(A|B|C) = \text{Digest}(\text{Digest}(\text{Digest}(\text{DiffID}(A) + " " + \text{DiffID}(B)) + " " + \text{DiffID}(C)))
\]

Hopefully, the above is illustrative of the actual contents of the ChainID. Most importantly, we can easily see that ChainID(C) ≠ ChainID(A|B|C), otherwise, ChainID(C) = DiffID(C), which is the base case,
could not be true.

ImageID

Each image's ID is given by the SHA256 hash of its configuration JSON. It is represented as a hexadecimal encoding of 256 bits, e.g., sha256:a9561eb1b190625c9adb5a9513e72c4dedafec1cb2d4c5236c9a6957ec7d7fd5a9. Since the configuration JSON that gets hashed references hashes of each layer in the image, this formulation of the ImageID makes images content-addressable.

Properties

Note: Any OPTIONAL field MAY also be set to null, which is equivalent to being absent.

- created string, OPTIONAL
  An combined date and time at which the image was created, formatted as defined by RFC 3339, section 5.6.

- author string, OPTIONAL
  Gives the name and/or email address of the person or entity which created and is responsible for maintaining the image.

- architecture string, REQUIRED
  The CPU architecture which the binaries in this image are built to run on. Configurations SHOULD use, and implementations SHOULD understand, values listed in the Go Language document for GOARCH.

- os string, REQUIRED
  The name of the operating system which the image is built to run on. Configurations SHOULD use, and implementations SHOULD understand, values listed in the Go Language document for GOOS.

- config object, OPTIONAL
  The execution parameters which SHOULD be used as a base when running a container using the image. This field can be null, in which case any execution parameters should be specified at creation of the container.

- User string, OPTIONAL
  The username or UID which is a platform-specific structure that allows specific control over which user the process run as. This acts as a default value to use when the value is not specified when creating a container. For Linux based systems, all of the following are valid: user, uid, user:group, uid:gid, uid:group:uid:gid. If group/gid is not specified, the default group and supplementary groups of the given user/uid in /etc/passwd from the container are applied.

- ExposedPorts object, OPTIONAL
  A set of ports to expose from a container running this image. Its keys can be in the format of: port/tcp, port/udp, port with the default protocol being tcp if not specified. These values act as defaults and are merged with any specified when creating a container. NOTE: This JSON structure value
is unusual because it is a direct JSON serialization of the Go type `map[string]struct{}` and is represented in JSON as an object mapping its keys to an empty object.

- **Env** *array of strings*, OPTIONAL
  
  Entries are in the format of `VARNAME=VARVALUE`. These values act as defaults and are merged with any specified when creating a container.

- **Entrypoint** *array of strings*, OPTIONAL
  
  A list of arguments to use as the command to execute when the container starts. These values act as defaults and may be replaced by an entrypoint specified when creating a container.

- **Cmd** *array of strings*, OPTIONAL
  
  Default arguments to the entrypoint of the container. These values act as defaults and may be replaced by any specified when creating a container. If an `Entrypoint` value is not specified, then the first entry of the `Cmd` array SHOULD be interpreted as the executable to run.

- **Volumes** *object*, OPTIONAL
  
  A set of directories describing where the process is likely write data specific to a container instance.

  NOTE: This JSON structure value is unusual because it is a direct JSON serialization of the Go type `map[string]struct{}` and is represented in JSON as an object mapping its keys to an empty object.

- **WorkingDir** *string*, OPTIONAL
  
  Sets the current working directory of the entrypoint process in the container. This value acts as a default and may be replaced by a working directory specified when creating a container.

- **Labels** *object*, OPTIONAL
  
  The field contains arbitrary metadata for the container. This property MUST use the annotation rules.

- **StopSignal** *string*, OPTIONAL
  
  The field contains the system call signal that will be sent to the container to exit. The signal can be a signal name in the format `SIGNAME`, for instance `SIGKILL` or `SIGRTMIN+3`.

- **rootfs** *object*, REQUIRED
  
  The `rootfs` key references the layer content addresses used by the image. This makes the image config hash depend on the filesystem hash.

  - **type** *string*, REQUIRED
    
    MUST be set to `layers`. Implementations MUST generate an error if they encounter a unknown value while verifying or unpacking an image.

  - **diff_ids** *array of strings*, REQUIRED
    
    An array of layer content hashes (DiffIDs), in order from first to last.

- **history** *array of objects*, OPTIONAL
Describes the history of each layer. The array is ordered from first to last. The object has the following fields:

- **created** `string`, OPTIONAL
  
  A combined date and time at which the layer was created, formatted as defined by RFC 3339, section 5.6.

- **author** `string`, OPTIONAL
  
  The author of the build point.

- **created_by** `string`, OPTIONAL
  
  The command which created the layer.

- **comment** `string`, OPTIONAL
  
  A custom message set when creating the layer.

- **empty_layer** `boolean`, OPTIONAL
  
  This field is used to mark if the history item created a filesystem diff. It is set to true if this history item doesn't correspond to an actual layer in the rootfs section (for example, Dockerfile's ENV command results in no change to the filesystem).

Any extra fields in the Image JSON struct are considered implementation specific and MUST be ignored by any implementations which are unable to interpret them.

Whitespace is OPTIONAL and implementations MAY have compact JSON with no whitespace.

**Example**

Here is an example image configuration JSON document:
"author": "Alyssa P. Hacker <alyspdev@example.com>",
"architecture": "amd64",
"os": "linux",
"config": {
  "User": "alice",
  "ExposedPorts": {
    "8080/tcp": {}
  },
  "Env": [
    "PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
    "FOO=oci_is_a",
    "BAR=well_written_spec"
  ],
  "Entrypoint": [
    "/bin/my-app-binary"
  ],
  "Cmd": [
    "--foreground",
    "--config",
    "/etc/my-app.d/default.cfg"
  ],
  "Volumes": {
    "/var/job-result-data": {},
    "/var/log/my-app-logs": {}
  },
  "WorkingDir": "/home/alice",
  "Labels": {
    "com.example.project.git.url": "https://example.com/project.git",
    "com.example.project.git.commit": "45a939b2999782a3f005621a8d0f29aa387e1d6b"
  }
},
"rootfs": {
  "diff_ids": [
    "sha256:c6f988f4874bb0add23a778f753c65efe992244e148a1d2ec2a8b664fb66bbd1",
    "sha256:5f70bf18a08607016e948b04aed3b82103a36bea41755b6cddfaf10ace3c6ef"
  ],
  "type": "layers"
},
"history": [
  {
    "created": "2015-10-31T22:22:54.690851953Z",
    "created_by": "/bin/sh -c #(nop) ADD file:a3bc1e842b69636f9df5256c49c5374fb4eef1e281fe3f282c65fb853ee171c5 in /
  },
  {
    "created": "2015-10-31T22:22:55.613815829Z",
    "created_by": "/bin/sh -c #(nop) CMD ["sh\"]",
    "empty_layer": true
  }
]
Appendices

© 2019 @vsocl All Rights Reserved
View on GitHub
The entity using a runtime to create a container MUST be able to use the operations defined in this specification against that same container. Whether other entities using the same, or other, instance of the runtime can see that container is out of scope of this specification.

State

The state of a container includes the following properties:

- **ociVersion** (string, REQUIRED) is version of the Open Container Initiative Runtime Specification with which the state complies.

- **id** (string, REQUIRED) is the container’s ID. This MUST be unique across all containers on this host. There is no requirement that it be unique across hosts.

- **status** (string, REQUIRED) is the runtime state of the container. The value MAY be one of:
  - creating: the container is being created (step 2 in the lifecycle)
  - created: the runtime has finished the create operation (after step 2 in the lifecycle), and the container process has neither exited nor executed the user-specified program
  - running: the container process has executed the user-specified program but has not exited (after step 5 in the lifecycle)
  - stopped: the container process has exited (step 7 in the lifecycle)

  Additional values MAY be defined by the runtime, however, they MUST be used to represent new runtime states not defined above.

- **pid** (int, REQUIRED when status is created or running on Linux, OPTIONAL on other platforms) is the ID of the container process, as seen by the host.

- **bundle** (string, REQUIRED) is the absolute path to the container’s bundle directory. This is provided so that consumers can find the container’s configuration and root filesystem on the host.

- **annotations** (map, OPTIONAL) contains the list of annotations associated with the container. If no annotations were provided then this property MAY either be absent or an empty map.

The state MAY include additional properties.

---

**Appendix C**

Excerpt of the **OCI** Runtime Spec [34].

The OpenContainers Runtime Spec

Runtime and Lifecycle

Scope of a Container

The entity using a runtime to create a container MUST be able to use the operations defined in this specification against that same container. Whether other entities using the same, or other, instance of the runtime can see that container is out of scope of this specification.

State

The state of a container includes the following properties:

- **ociVersion** (string, REQUIRED) is version of the Open Container Initiative Runtime Specification with which the state complies.

- **id** (string, REQUIRED) is the container’s ID. This MUST be unique across all containers on this host. There is no requirement that it be unique across hosts.

- **status** (string, REQUIRED) is the runtime state of the container. The value MAY be one of:
  - creating: the container is being created (step 2 in the lifecycle)
  - created: the runtime has finished the create operation (after step 2 in the lifecycle), and the container process has neither exited nor executed the user-specified program
  - running: the container process has executed the user-specified program but has not exited (after step 5 in the lifecycle)
  - stopped: the container process has exited (step 7 in the lifecycle)

  Additional values MAY be defined by the runtime, however, they MUST be used to represent new runtime states not defined above.

- **pid** (int, REQUIRED when status is created or running on Linux, OPTIONAL on other platforms) is the ID of the container process, as seen by the host.

- **bundle** (string, REQUIRED) is the absolute path to the container’s bundle directory. This is provided so that consumers can find the container’s configuration and root filesystem on the host.

- **annotations** (map, OPTIONAL) contains the list of annotations associated with the container. If no annotations were provided then this property MAY either be absent or an empty map.

The state MAY include additional properties.
When serialized in JSON, the format MUST adhere to the following pattern:

```json
{
    "ociVersion": "0.2.0",
    "id": "oci-container1",
    "status": "running",
    "pid": 4422,
    "bundle": "/containers/redis",
    "annotations": {
        "myKey": "myValue"
    }
}
```

See [Query State](#) for information on retrieving the state of a container.

**Lifecycle**

The lifecycle describes the timeline of events that happen from when a container is created to when it ceases to exist.

1. OCI compliant runtime's `create` command is invoked with a reference to the location of the bundle and a unique identifier.
2. The container's runtime environment MUST be created according to the configuration in `config.json`. If the runtime is unable to create the environment specified in the `config.json`, it MUST generate an error. While the resources requested in the `config.json` must be created, the user-specified program (from `process`) MUST NOT be run at this time. Any updates to `config.json` after this step MUST NOT affect the container.
3. The `prestart hooks` MUST be invoked by the runtime. If any `prestart` hook fails, the runtime MUST generate an error, stop the container, and continue the lifecycle at step 12.
4. The `createRuntime hooks` MUST be invoked by the runtime. If any `createRuntime` hook fails, the runtime MUST generate an error, stop the container, and continue the lifecycle at step 12.
5. The `createContainer hooks` MUST be invoked by the runtime. If any `createContainer` hook fails, the runtime MUST generate an error, stop the container, and continue the lifecycle at step 12.
6. Runtime's `start` command is invoked with the unique identifier of the container.
7. The `startContainer hooks` MUST be invoked by the runtime. If any `startContainer` hook fails, the runtime MUST generate an error, stop the container, and continue the lifecycle at step 12.
8. The runtime MUST run the user-specified program, as specified by `process`.
9. The `poststart hooks` MUST be invoked by the runtime. If any `poststart` hook fails, the runtime MUST log a warning, but the remaining hooks and lifecycle continue as if the hook had succeeded.
10. The container process exits. This MAY happen due to erroring out, exiting, crashing or the runtime's `kill` operation being invoked.
11. Runtime's `delete` command is invoked with the unique identifier of the container.
12. The container MUST be destroyed by undoing the steps performed during create phase (step 2).
13. The `poststop hooks` MUST be invoked by the runtime. If any `poststop` hook fails, the runtime MUST log a warning, but the remaining hooks and lifecycle continue as if the hook had succeeded.

**Errors**
In cases where the specified operation generates an error, this specification does not mandate how, or even if, that error is returned or exposed to the user of an implementation. Unless otherwise stated, generating an error MUST leave the state of the environment as if the operation were never attempted - modulo any possible trivial ancillary changes such as logging.

**Warnings**

In cases where the specified operation logs a warning, this specification does not mandate how, or even if, that warning is returned or exposed to the user of an implementation. Unless otherwise stated, logging a warning does not change the flow of the operation; it MUST continue as if the warning had not been logged.

**Operations**

Unless otherwise stated, runtimes MUST support the following operations.

Note: these operations are not specifying any command-line APIs, and the parameters are inputs for general operations.

**Query State**

```
state <container-id>
```

This operation MUST generate an error if it is not provided the ID of a container. Attempting to query a container that does not exist MUST generate an error. This operation MUST return the state of a container as specified in the State section.

**Create**

```
create <container-id> <path-to-bundle>
```

This operation MUST generate an error if it is not provided a path to the bundle and the container ID to associate with the container. If the ID provided is not unique across all containers within the scope of the runtime, or is not valid in any other way, the implementation MUST generate an error and a new container MUST NOT be created. This operation MUST create a new container.

All of the properties configured in config.json except for process MUST be applied. process.args MUST NOT be applied until triggered by the start operation. The remaining process properties MAY be applied by this operation. If the runtime cannot apply a property as specified in the configuration, it MUST generate an error and a new container MUST NOT be created.

The runtime MAY validate config.json against this spec, either generically or with respect to the local system capabilities, before creating the container (step 2). Runtime callers who are interested in pre-create validation can run bundle-validation tools before invoking the create operation.

Any changes made to the config.json file after this operation will not have an effect on the container.

**Start**

```
start <container-id>
```

This operation MUST generate an error if it is not provided the container ID. Attempting to start a
container that is not **created** MUST have no effect on the container and MUST **generate an error**. This operation MUST run the user-specified program as specified by **process**. This operation MUST generate an error if **process** was not set.

**Kill**

```plaintext
kill <container-id> <signal>
```

This operation MUST **generate an error** if it is not provided the container ID. Attempting to send a signal to a container that is neither **created** nor **running** MUST have no effect on the container and MUST **generate an error**. This operation MUST send the specified signal to the container process.

**Delete**

```plaintext
delete <container-id>
```

This operation MUST **generate an error** if it is not provided the container ID. Attempting to delete a container that is not **stopped** MUST have no effect on the container and MUST **generate an error**. Deleting a container MUST delete the resources that were created during the **create** step. Note that resources associated with the container, but not created by this container, MUST NOT be deleted. Once a container is deleted its ID MAY be used by a subsequent container.

**Hooks**

Many of the operations specified in this specification have "hooks" that allow for additional actions to be taken before or after each operation. See **runtime configuration for hooks** for more information.