Distributed Execution of Unikernel Applications on Container Orchestrator Platform Kubernetes for IoT Scenarios

Author: **Atakan Yenel**
Supervisor: Prof. Dr. Michael Gerndt
Advisor: Vladimir Podolskiy

Introduction

- Run very small virtual machines on hypervisors and IoT devices
- Orchestrate them from the cloud
- Use them side by side with the cloud ecosystem

Motivation

- Security
- Resource utilization
- Scaling time
- IoT scenarios

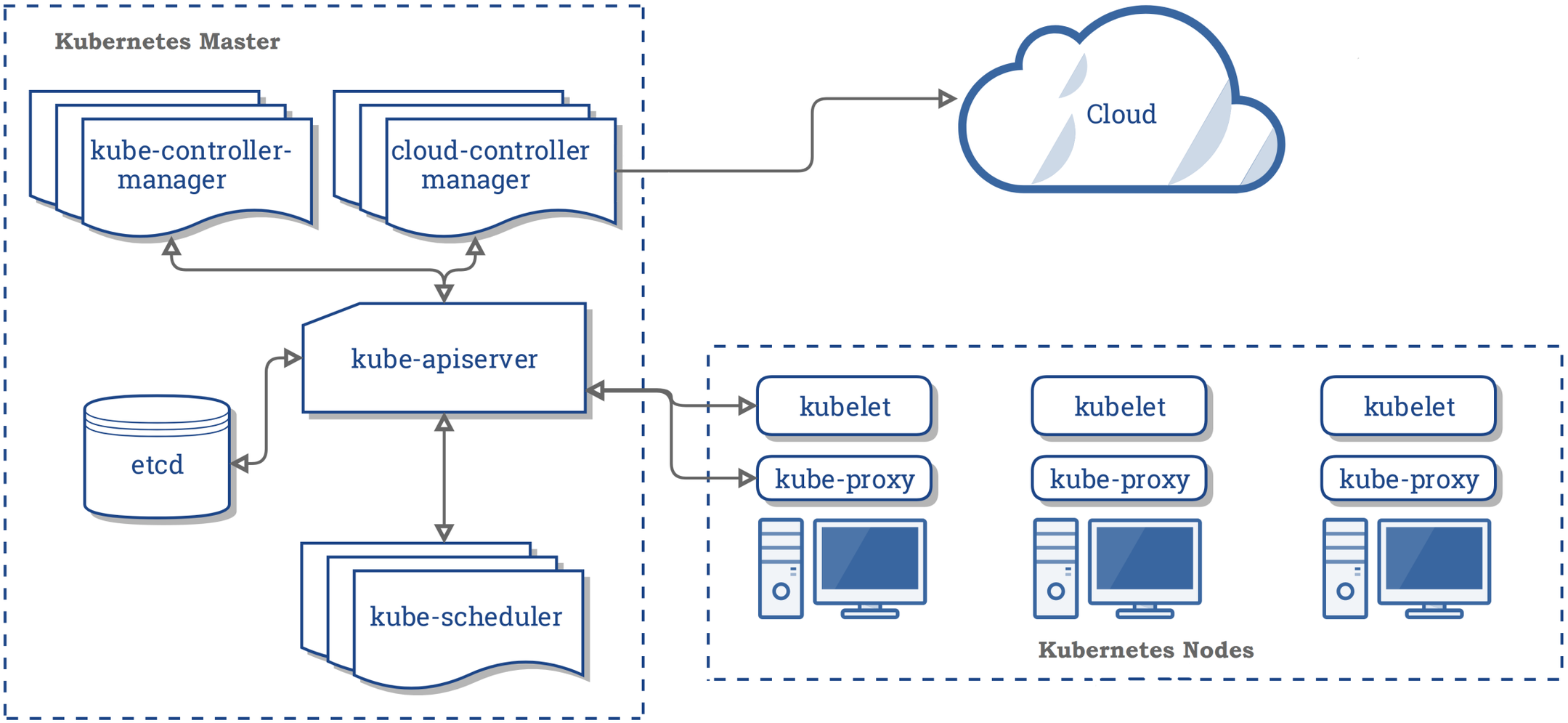
Background

- Kubernetes
- Unikernel

Kubernetes

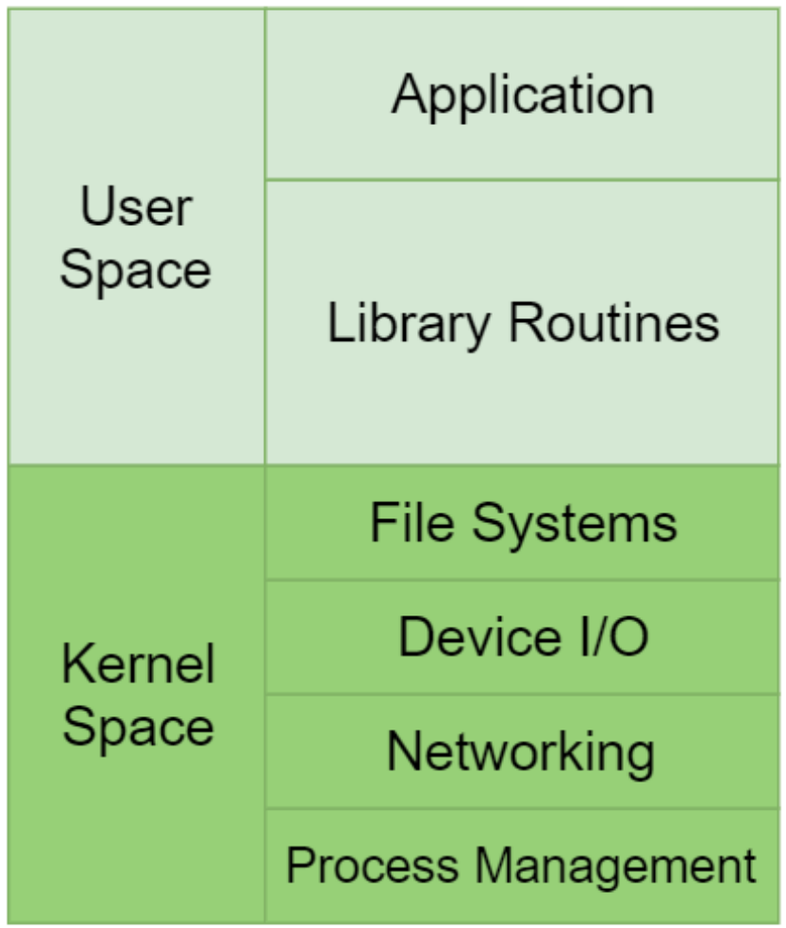
- Manages containerized applications across multiple hosts
- Developed & open sourced by Google
- Deployment & maintenance & **scaling** of applications
- Smallest deployable unit is pod, a group of containers
- Allows labeling hosts (nodes)

Kubernetes Architecture

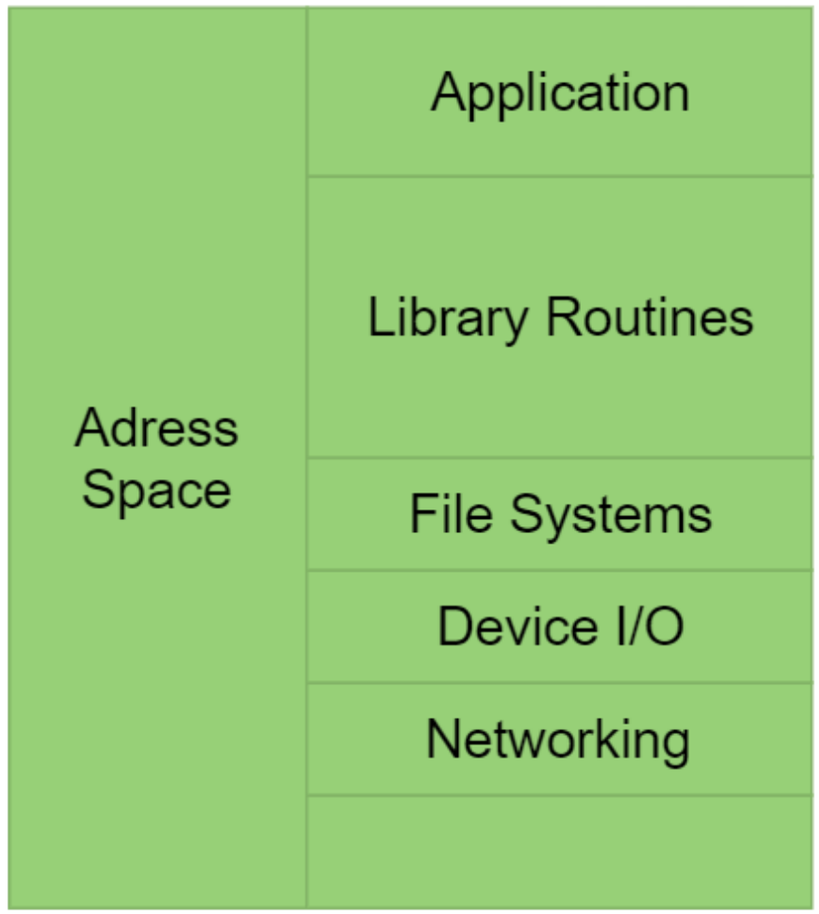


Unikernel

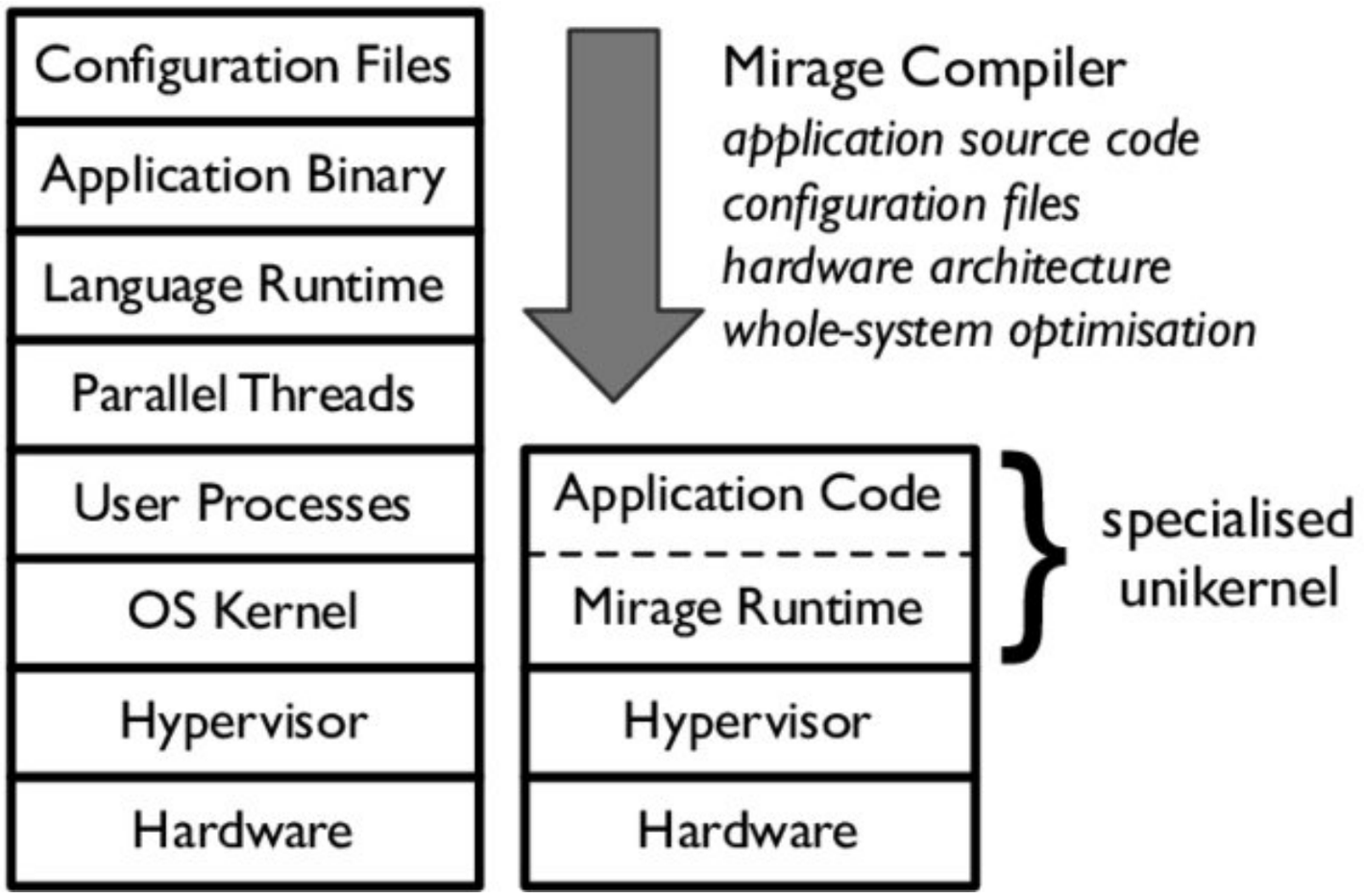
- Specialized, single address space machine images
- Minimal operating system for running the application code
- Runs directly on hypervisors & hardware
- First such system is Exokernel



Normal Application Stack



Unikernel Application Stack

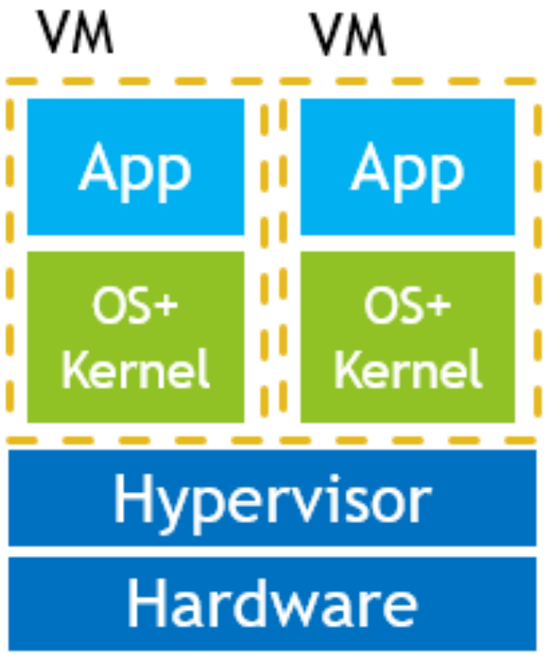

Configuration Files
Application Binary
Language Runtime
Parallel Threads
User Processes
OS Kernel
Hypervisor
Hardware
Mirage Compiler
application source code
configuration files
hardware architecture
whole-system optimisation
Application Code
Mirage Runtime
Hypervisor
Hardware
specialised
unikernel

Mirage OS

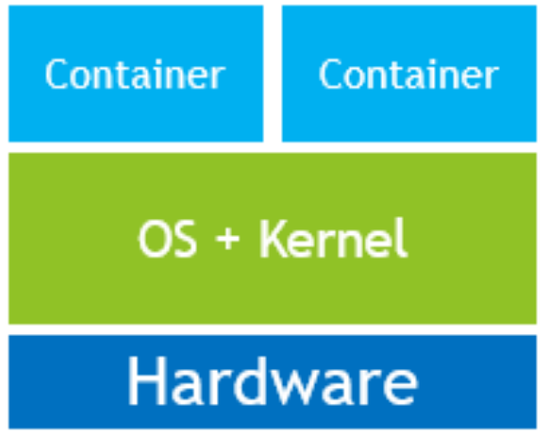
- Developed with OCaml
- From University of Cambridge & open sourced
- Runs on Xen & KVM
- `mirage configure -t xen && make depend && mirage build`
- Provides libraries for:
    - Networking
    - Storage
    - Concurrency Support
- When compiled, libraries become OS drivers

Mirage OS

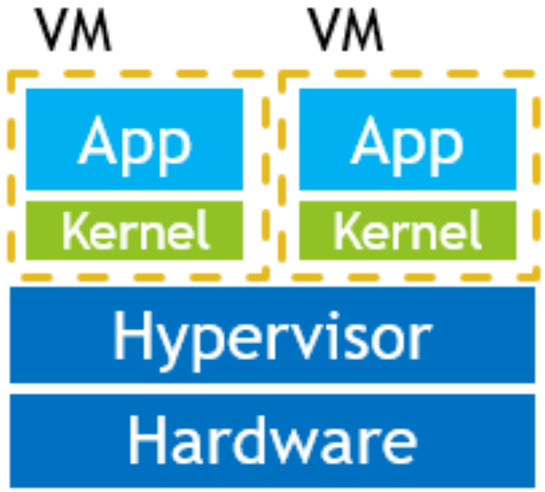
```
open Lwt.Infix

module Main (KV: Mirage_kv.RO) (Time: Mirage_time.S) = struct

  let start kv _time=

    let read_from_file kv filename =
        KV.get kv (Mirage_kv.Key.v filename) >|= function
            | Error e ->
                Logs.warn (fun f -> f "Cannot find the file %a"
                KV.pp_error e)
            | Ok sensor_value ->
                Logs.info (fun f -> f "Reading from: %s -> %s" filename sensor_value);
    in
        let filename=Key_gen.filename() in
        let rec loop() =
        read_from_file kv filename >>= fun()->
        Time.sleep_ns (Duration.of_sec 2)>>= fun () ->
        loop()
        in
        loop()
end
```



Virtual Machines

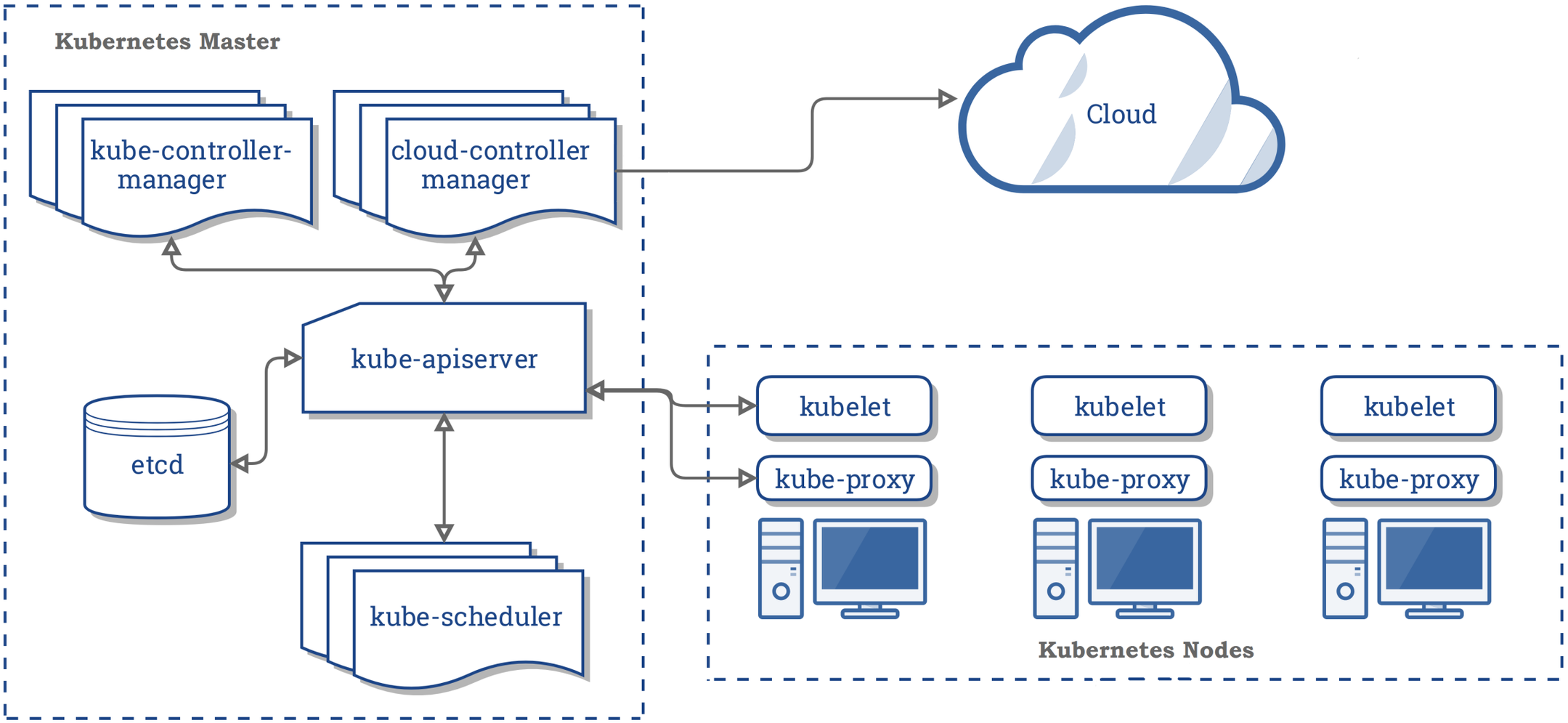


Linux Containers

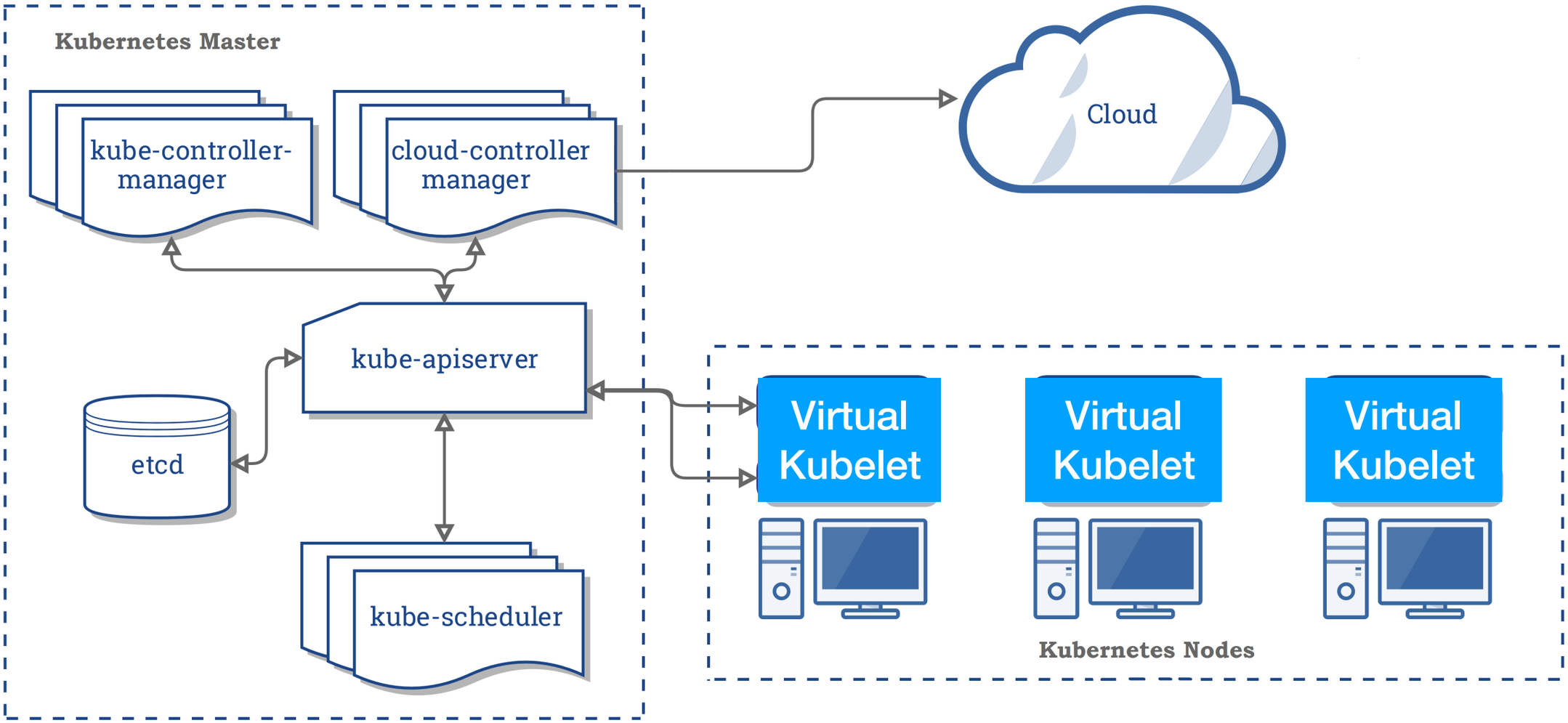


Unikernels

Implementation

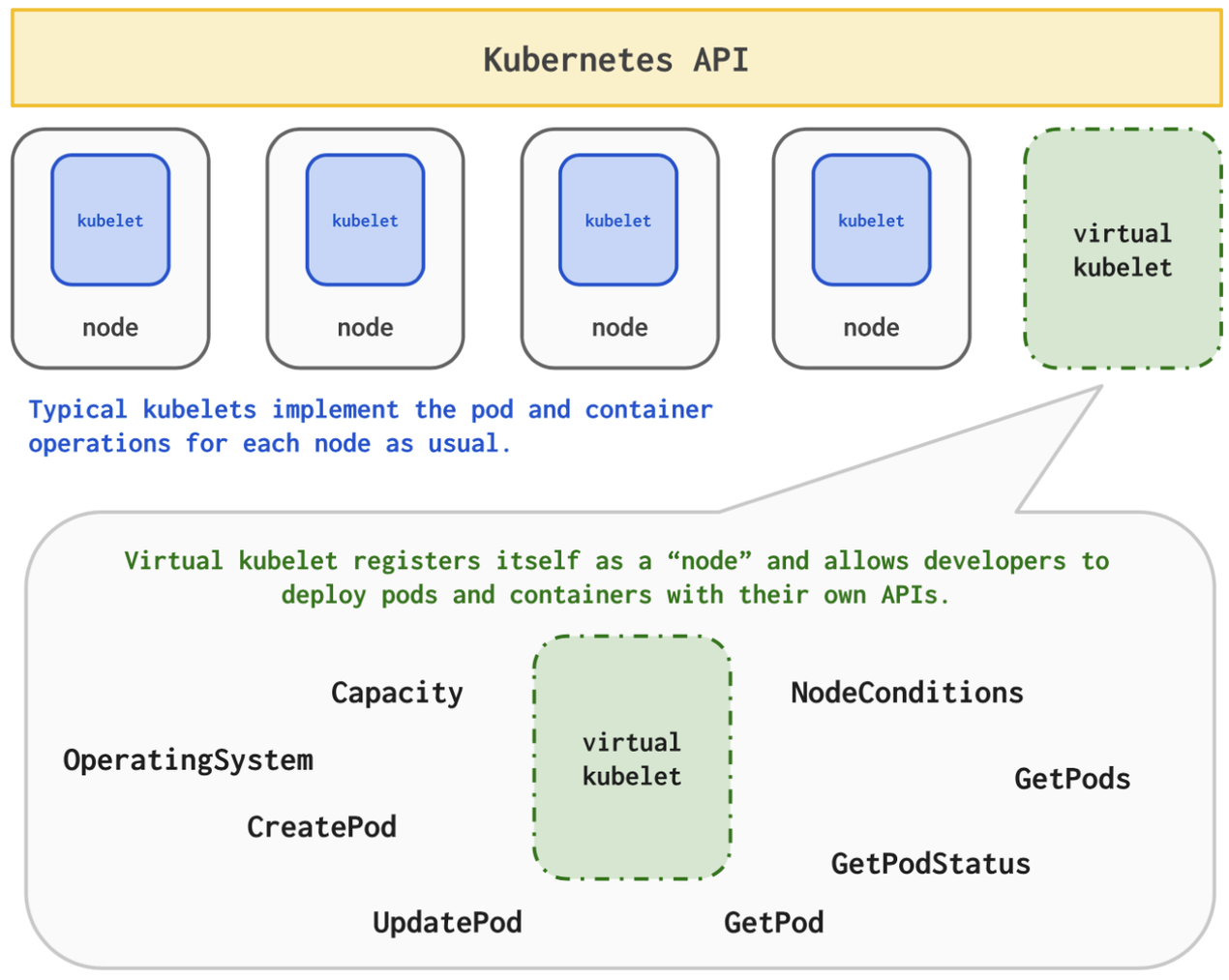


Implementation



Virtual-Kubelet

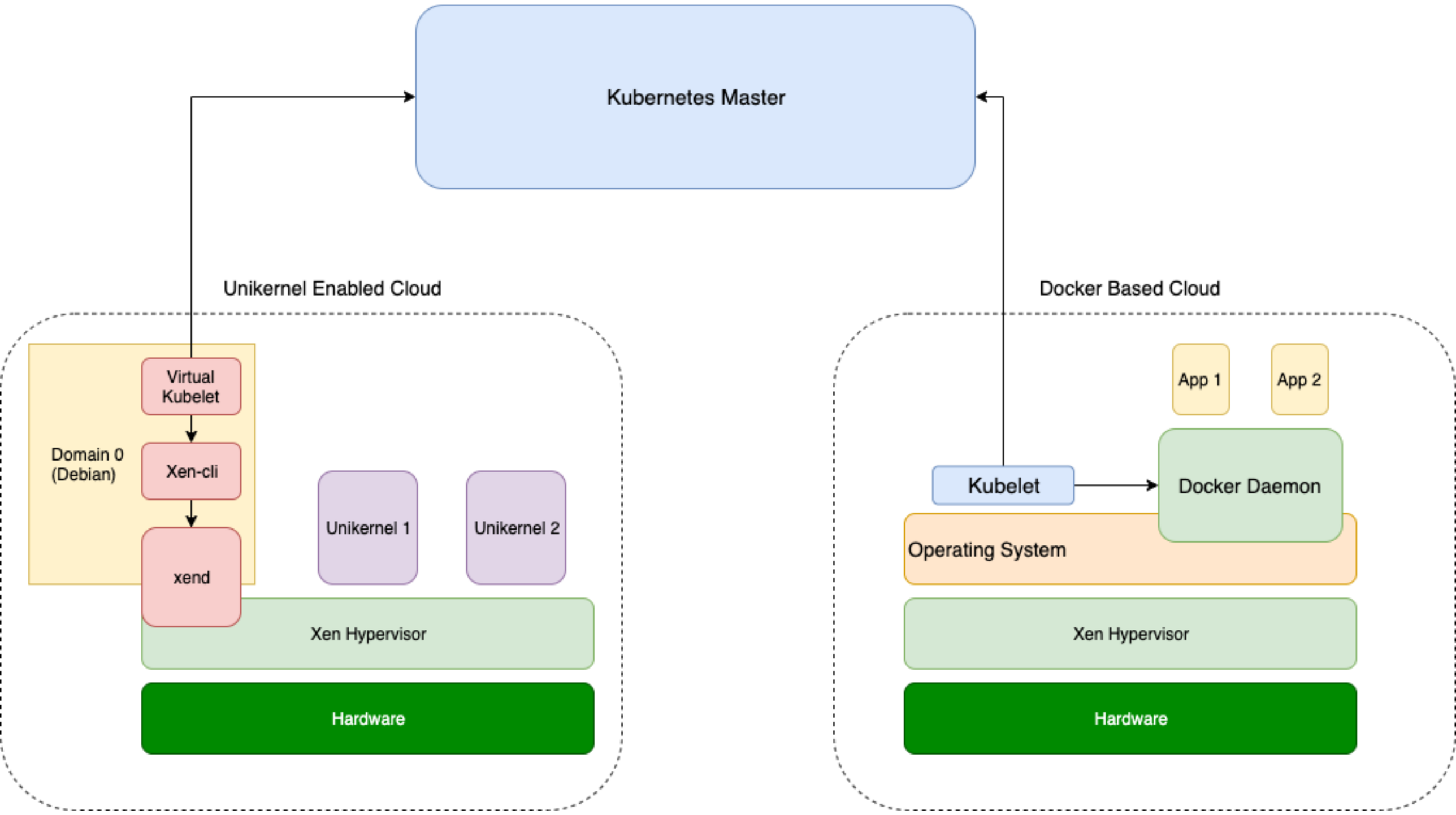
- A Kubelet Facade
- Developed by Microsoft
- Has an extensible API
- 36 MB for linux



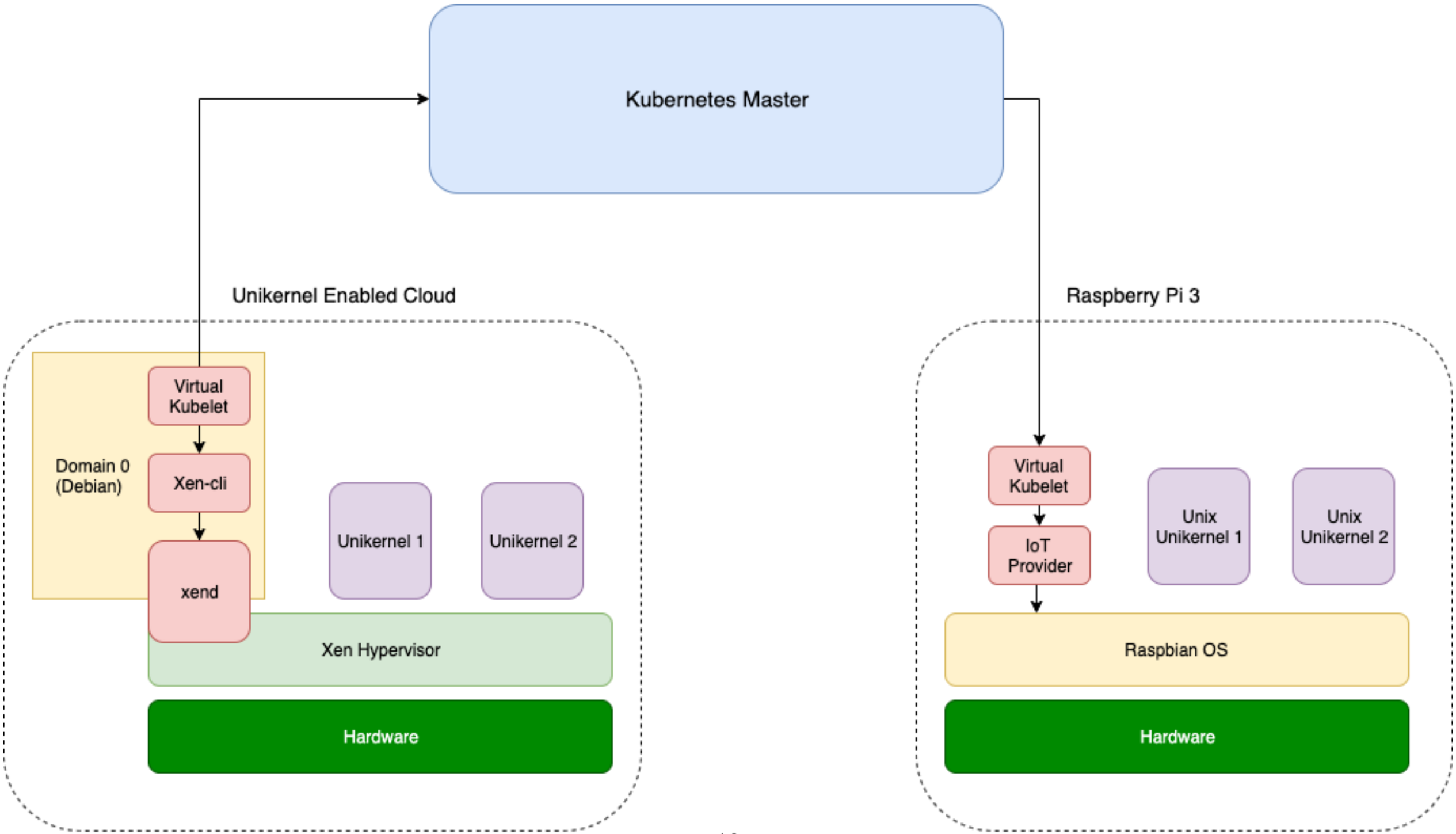
Kubelet API

```
type PodLifecycleHandler interface {

    CreatePod(ctx context.Context, pod *corev1.Pod) error

    UpdatePod(ctx context.Context, pod *corev1.Pod) error

    DeletePod(ctx context.Context, pod *corev1.Pod) error

    GetPod(ctx context.Context, namespace, name string) (*corev1.Pod, error)

    GetPodStatus(ctx context.Context, namespace, name string) (*corev1.PodStatus, error)

    GetPods(context.Context) ([]*corev1.Pod, error)
}
```

Architecture



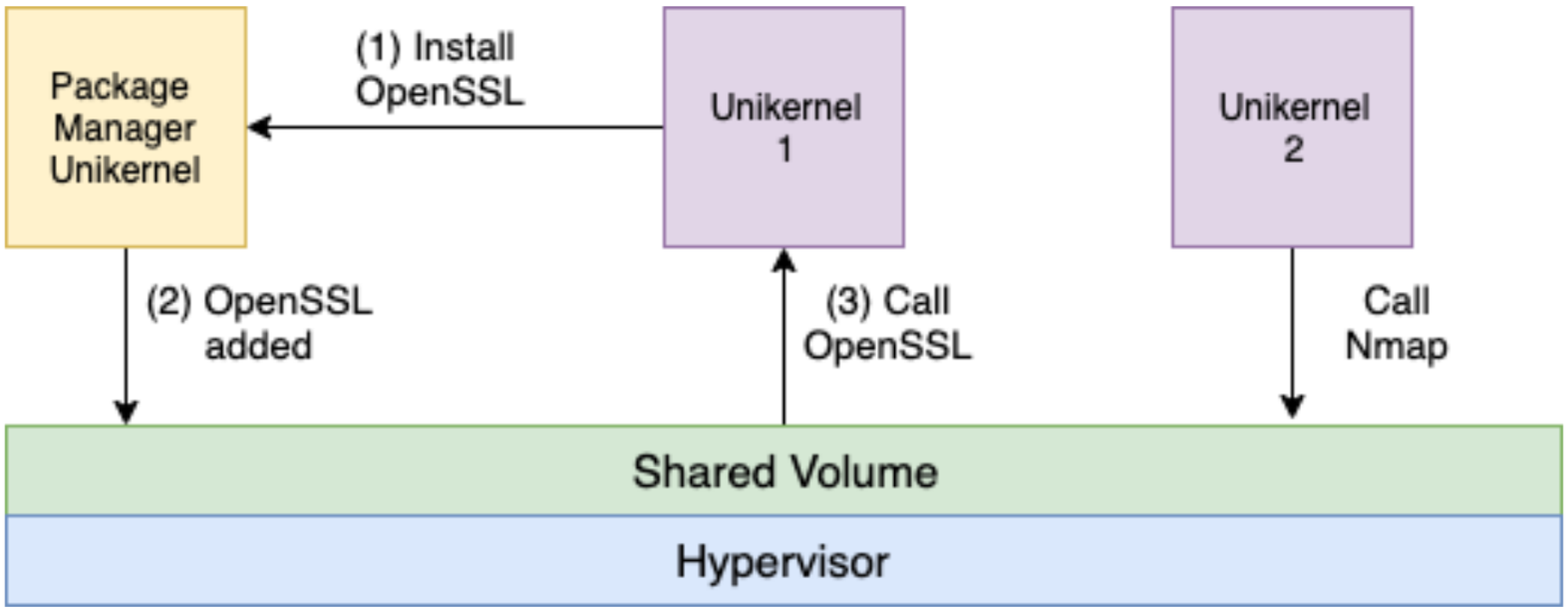
Managing IoT



Label-based deployment

```
apiVersion: apps/v1
kind: DaemonSet
metadata:
  name: green-daemon
  namespace: default

spec:
 (***)
    spec:
      containers:
        - name: uni-run
    image: humidity-daemon
    args:
      - GREEN
nodeSelector:
    type: virtual-kubelet
    location: germany
    sensor: humidity
    os: raspbian
tolerations:
- key: virtual-kubelet.io/provider
  operator: Equal
  value: unikernel
  effect: NoSchedule
```

Calling external packages

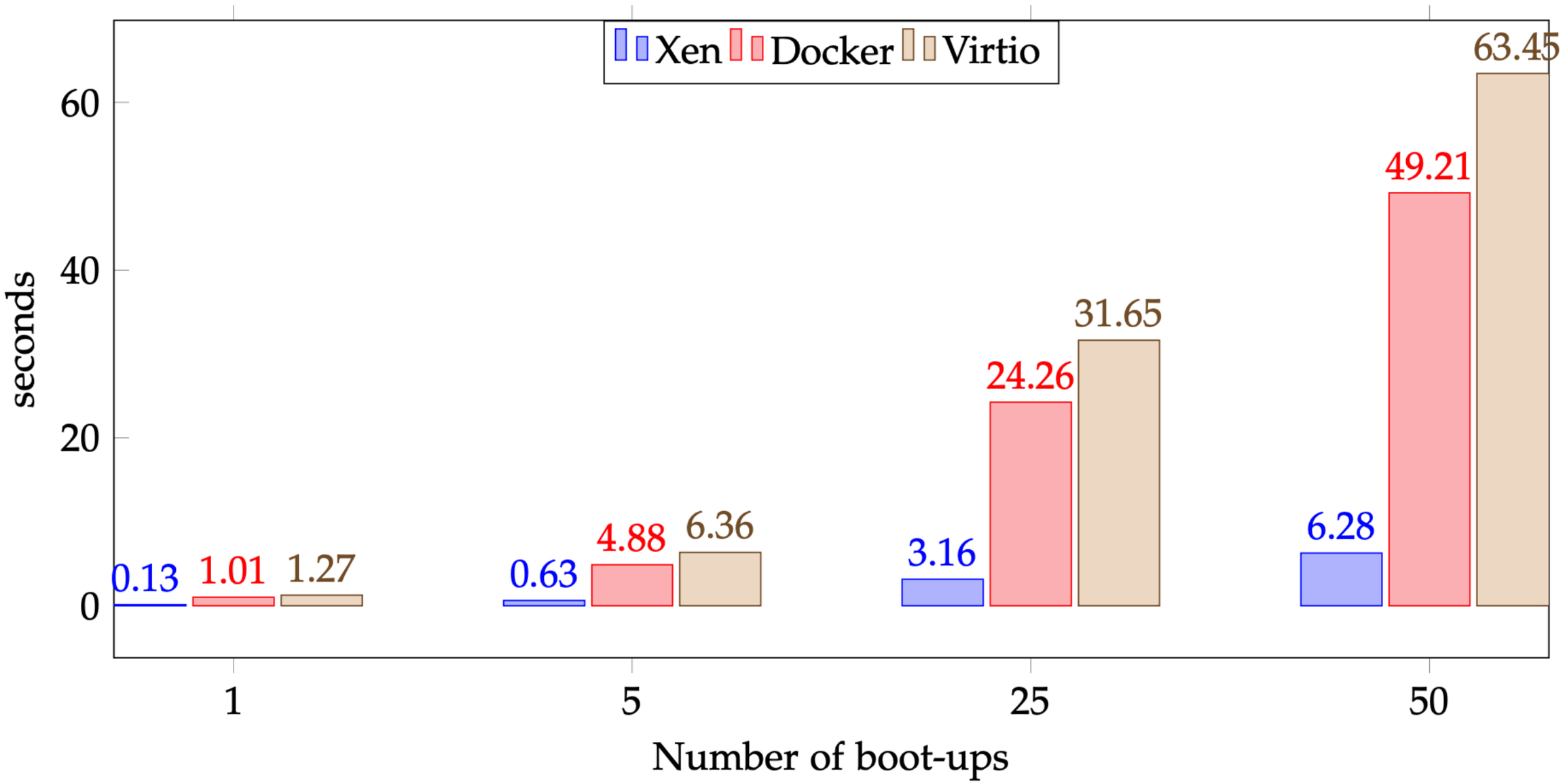


Evaluation

- Image Size
- Boot up
- Scalability
- Usability

Target	Size
Docker	69.4 MB
Unix	4.7 MB
Xen	3.3 MB
Virtio	2.4 MB
Hvt	2.3 MB

Scalability



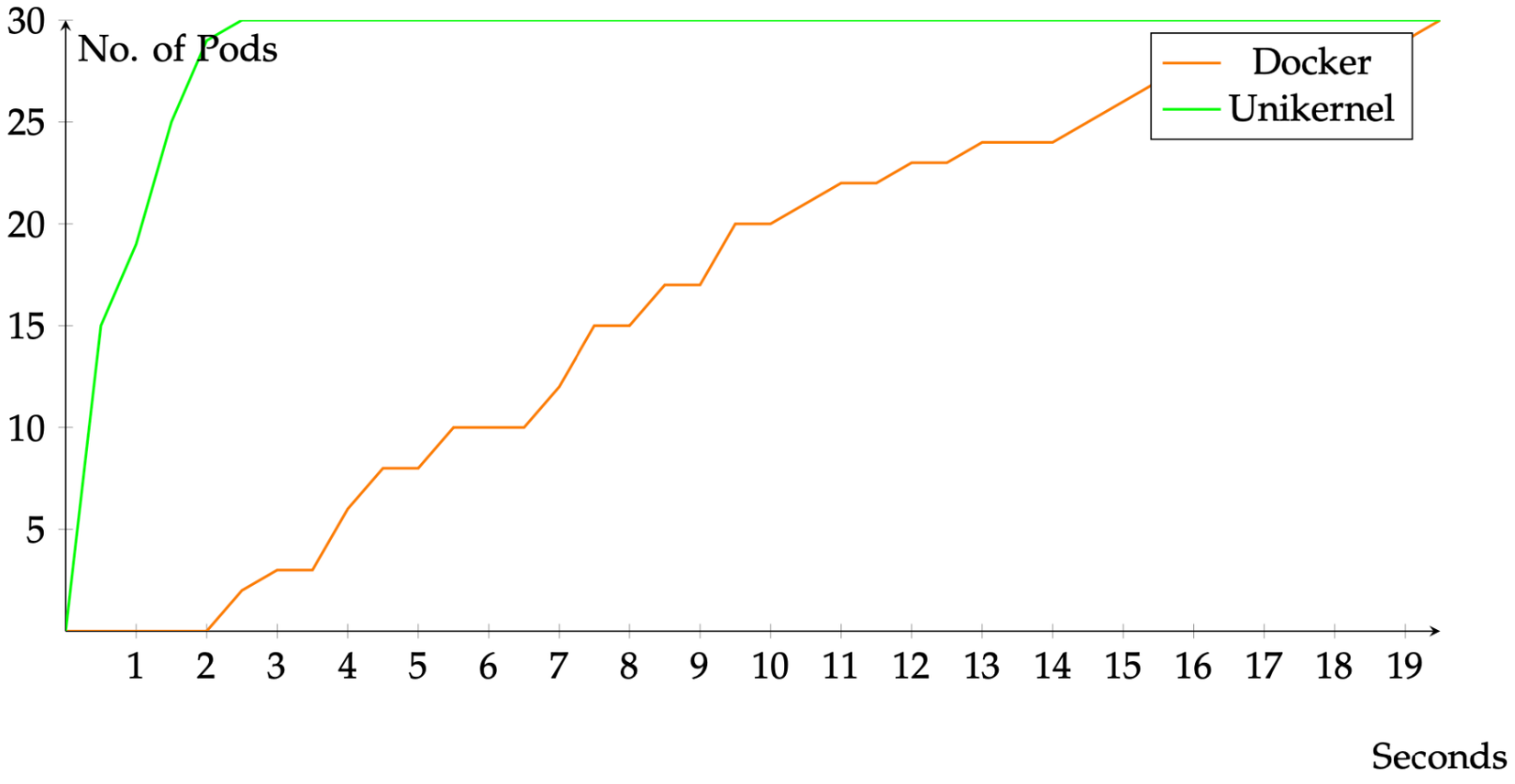


Figure 5.5.: Scaling from 0 to 30 pods

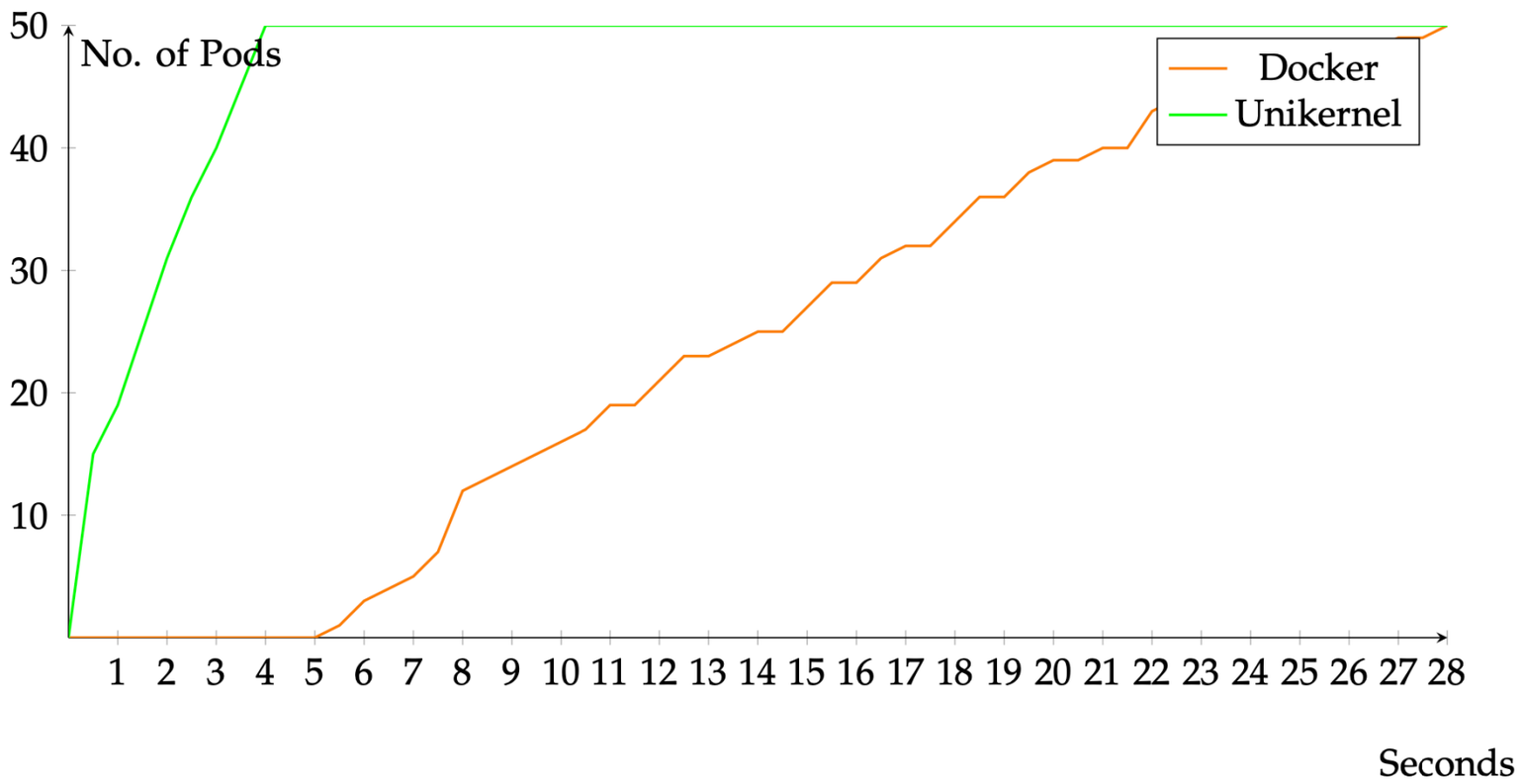


Figure 5.6.: Scaling from 0 to 50 pods

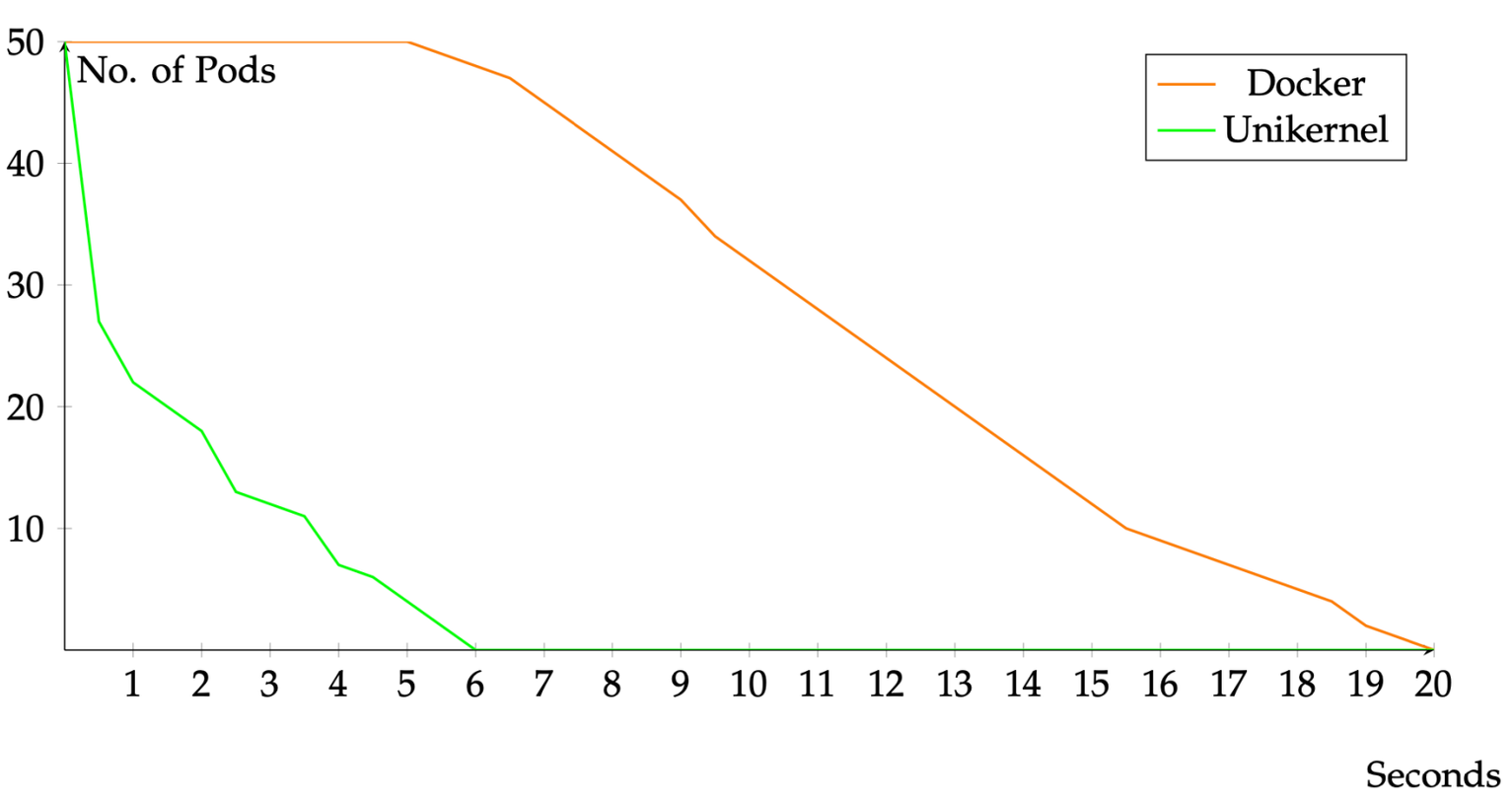


Figure 5.7.: Scaling from 50 to 0 pods

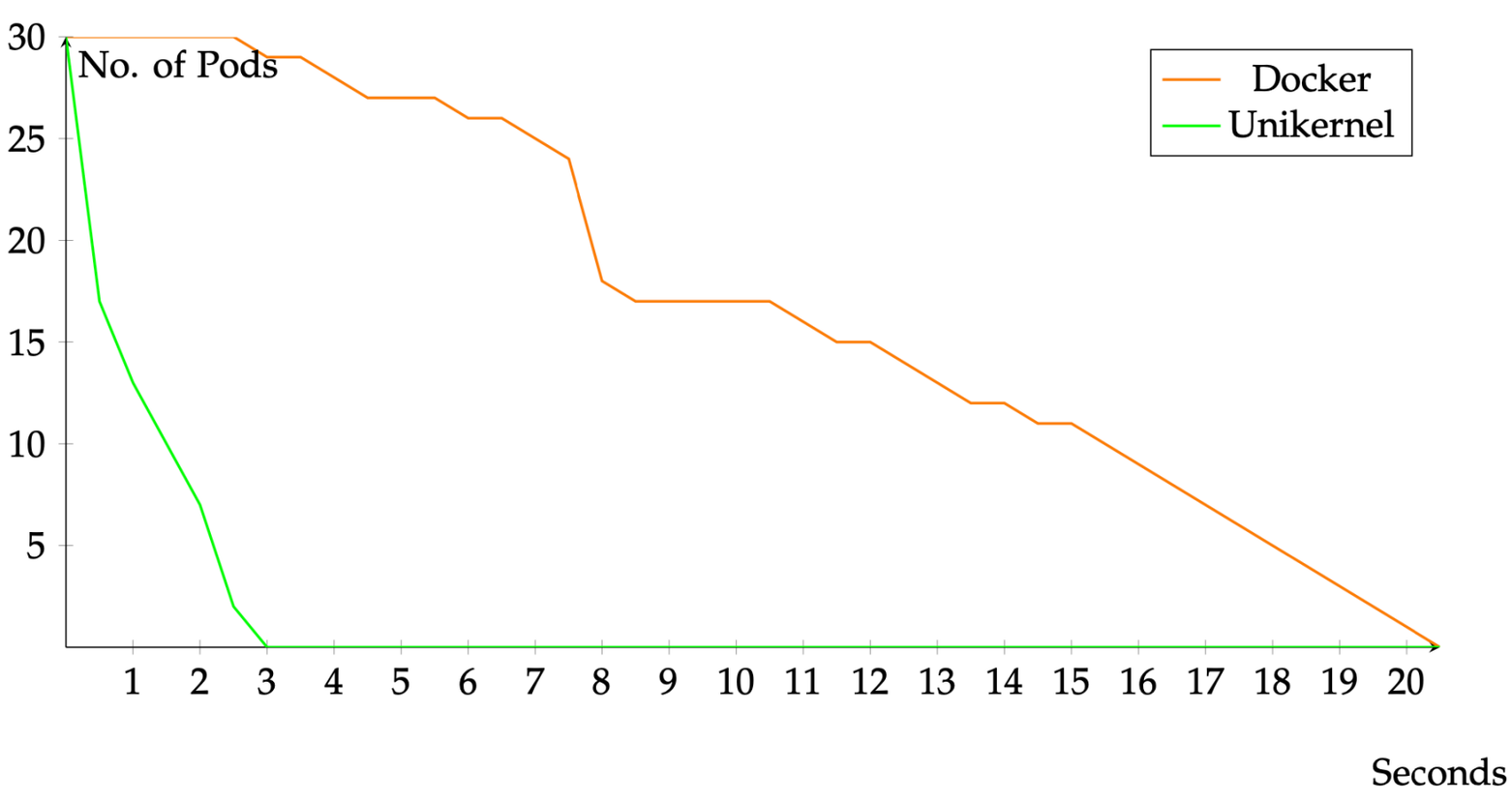


Figure 5.8.: Scaling from 30 to 0 pods

Motivation revisited

- **Security**
  - Single address space
  - Only process is application
  - Isolated by hypervisor
- **Resource utilization**
  - Runs on hypervisor
  - No background processes
- **Scaling Time**
  - Faster than Docker
  - Smaller image size
- **IoT Scenarios**
  - Single interface for cloud & IoT
  - DaemonSet support
  - Kubernetes

Drawbacks

- Very limited language support for Unikernels
- Not all IoT devices support virtualization
- Kubernetes uses HTTP, additional overhead for IoT
- Docker is good enough
    - Has dependencies installed

Contributions

- Unikernel images:
    - For reading sensor data
    - For calling external packages
- A way to create large-scale virtual clusters
- A hybrid Kubernetes cluster with cloud VMs, local VMs and IoT devices
- A Kubernetes application for managing virtual nodes (nodeWatcher)
- Virtual-kubelet fork that allows labeling at start
- Kubelet implementations:
    - A provider for managing Xen virtual machines
    - A provider for managing unix processes on Raspberry Pi

References

- A. Yenel. “Distributed Execution of Unikernel Applications on Container Orchestration Platform Kubernetes for IoT Scenarios”. Masterarbeit. Technische Universität München, 2020. (mediaTUM)