Demystifying container networking

Jasu Viding

HELSINGIN YLIOPISTO — HELSINGFORS UNIVERSITET — UNIVERSITY OF HELSINKI

Tiedekunta — Fakultet — Faculty	Laitos — Institution — Department
Faculty of Science	Department of Computer Science

Tekijä — Författare — Author
Jasu Viding

Työn nimi — Arbetets titel — Title
Demystifying container networking

Oppiaine — Läroämne — Subject
Computer Science

Työn laji — Arbetets art — Level	Aika — Datum — Month and year	Sivumäärä — Sidoantal — Number of pages
	August 30, 2020	68 pages + 3 appendix pages

Tiivistelmä — Referat — Abstract

A cluster of containerized workloads is a complex system where stacked layers of plugins and interfaces can quickly hide what's actually going on under the hood. This can result in incorrect assumptions, security incidents, and other disasters. With a networking viewpoint, this paper dives into the Linux networking subsystem to demystify how container networks are built on Linux systems. This knowledge of "how" allows then to understand the different networking features of Kubernetes, Docker, or any other containerization solution developed in the future.

ACM Computing Classification System (CCS):
General and reference → Document types → Surveys and overviews
Applied computing → Document management and text processing → Document management → Text editing

Avainsanat — Nyckelord — Keywords
ulkoasu, tiivistelmä, lähdeluettelo

Säilytyspaikka — Förvaringsställe — Where deposited

Muita tietoja — övriga uppgifter — Additional information

Contents

1   Introduction

Since the inception of Docker in 2013 and the announcement of Kubernetes project in 2014, containers have become one of the fastest growing technologies in information technology. Organizations are adopting them at an increasing rate while also reporting issues with their complexity. A cluster of containerized workloads is indeed a complex system where the stacked layers of plugins and interfaces can quickly hide what really is going on under the hood.

Figuring out the internals of Kubernetes and Docker, and how they actually operate is a challenge. Being still fairly new technology they are subject to continuous change, and different documentation, blog posts, and guides quickly become outdated. Most of these also have a tendency to describe "what" instead of "how" without going deep down to the internals.

In this paper, these technologies are analyzed from the networking viewpoint. The paper dives deep into the Linux networking subsystem, then connects it with container technology, and shows how container networks are actually built. The aim is to demystify container networking by describing and interconnecting the different layers and interfaces related to it, and accompany these descriptions with practical examples run on the Linux terminal.

Compared to most Kubernetes or Docker literature, this paper is more comprehensive in terms of networking. Instead of "what", this paper answers the question "how" by approaching container networking through the Linux networking subsystem that is the foundation of all the networking on a Linux system. With this knowledge of "how", the networking features of Kubernetes or Docker become easy to understand as well as of any container solution in the future.

The remainder of this paper is structured as follows. Section 2 describes container technology in detail and introduces the Linux networking subsystem. In section 2, bridge mode and overlay mode container networks are set up from scratch with Linux network devices. Section 4 describes container networking in a Kubernetes cluster. Section 5 discusses some challenges related to container networking and container technology in general. Section 6 encloses the paper with conclusions.

1.1   Typographic conventions

This paper contains some practical examples with commands executed in the Linux terminal. The presentation follows the following conventions:

```
# This is a comment
$ This is a command to be executed
$> This is a command continued on a new line
> This is output from an executed command
```

2 State of the art

Containers are a lightweight alternative to virtual machines and can be used to run multiple Linux systems on a single host. One of the main benefits of containers over virtual machines is their near native performance. They are also fast to create, distribute, run, and destroy.

In Linux, containers are not "Linux container primitives" but groups of processes that share the same underlying hardware and kernel. Each process is associated with common Linux tools to create it an isolated environment that appears to the Linux system running inside the container as if it was the host system itself. The most important of these tools are cgroups and namespaces.

Docker, the most popular container engine, is a project that aims in making it easy and safe to create and use containers. It's not only a container runtime, but a platform to manage the entire life cycle of containers. Their tools have also brought standardization to container technologies.

The Linux network stack is in the core of Linux networking. It's what enables networking in a Linux system, and usually containers each have their own copy of the Linux network stack for network connectivity. The stack has seven layers and its design follows the guidelines set in the OSI model and TCP/IP protocol suite.

The Linux network stack is supported by a number of network devices that can be used to build different container networks. These devices are part of the Linux kernel networking subsystem that, in fact, comprises most of the Linux kernel code. Understanding this subsystem and its devices is essential when building container networks.

2.1 Virtual machines

Virtual machine [1] is a virtualized system run on top of another system. It comes with its own virtual hardware, including CPUs, memory, hard drives, networking interfaces, and other devices. These virtual hardware create a layer of abstraction between the virtual machine and the underlying host system.

A virtual machine can run an operating system different from its host, and multiple virtual machines can be run on a single host simultaneously (Figure 1). Virtual machines are also well isolated from one another and from the host. To an application running inside a virtual machine, it appears as if it was running inside its own dedicated host system.

One of the main benefits in running multiple virtual machines on a single host system is more efficient resource usage and better scalability. Instead of deploying multiple servers, each operating at a fraction of their capacity, multiple virtual machines can be deployed on a single server. Virtual machines are also easy to create, start, pause, move on a new host, resume, stop, and destroy.

A downside of virtual machines is that they have a tendency to consume their
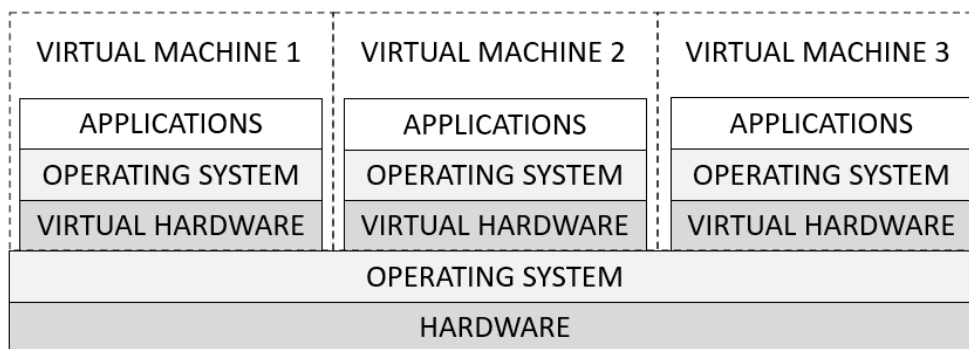
Figure 1: Multiple virtual machines on a single host system.

entire allowance of resource usage even when idle. Virtual machines run their own operating systems, and an operating system is capable of keeping itself busy even when there's no user activity. As a result, resources are not always used most efficiently and can go to waste [2]. In the other hand, the resource consumption is predictable, and situations where a virtual machine would exhaust other virtual machines running on the same host can be avoided.

2.2 Linux containers

Linux containers are a form of operating-system-level virtualization. Unlike virtual machines, that come with their own virtual hardware, containers use the kernel and hardware of the underlying host system. This makes them a more lightweight alternative to virtual machines.

One of the most common Linux container implementations is the LXC. It combines a number of Linux kernel features for virtualization, isolation, and resource usage [3]. The most important of these features are *cgroups* and *namespaces*. Cgroups, also known as control groups, are a Linux kernel feature to limit and prioritize resource usage [4]. Namespaces are isolation functionalities that allow complete isolation of an application's view of the operating environment [5].

Just like virtual machines, multiple Linux containers can be run simultaneously on a single Linux host (Figure 2). Because they all share the same kernel and hardware, they can reach near native performance. In the other hand, the downside of the shared kernel is that only Linux containers can be run on a Linux system. So, for example, a Windows container can't be run on a Linux system.

Even that containers can be superior to virtual machines in some aspects, they are not flawless. The shared kernel and hardware reduce container isolation, and it's not unheard of that a process breaks out from a container gaining access to the host system [6]. Also, such as usage spikes on a containerized process may exhaust other containers running on the same host.
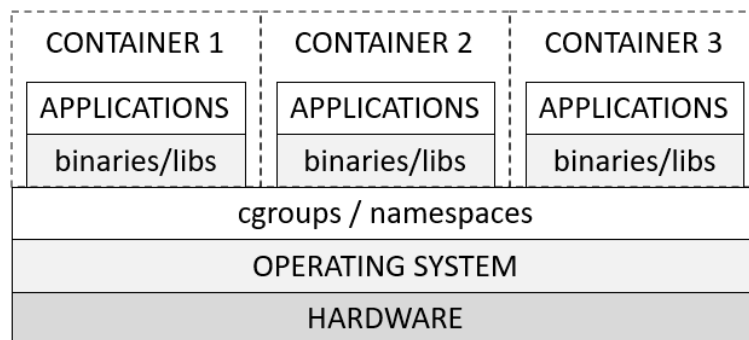
Figure 2: Linux containers on a single host.

2.2.1   Operating system

To understand Linux containers better, it's good to understand what is a Linux operating system. Linux as itself is not an operating system but a kernel. Kernel is an interface between software and a computer's hardware. It controls all the major functions of the hardware including memory management, process management, and device drivers.

The kernel works in its own little world known as kernel space that is invisible to the users. It manages all the user processes that are run in another world known as user space [7]. The kernel's services can be requested from the user space through a set of program interfaces known as system calls.

Linux operating systems such as Debian, Ubuntu, and CentOS are different package distributions that are wrapped around the Linux kernel. They each include different software such as package managers, display servers, and other extra tools. But regardless of their differences, all Linux operating systems share the same Linux kernel in their core.
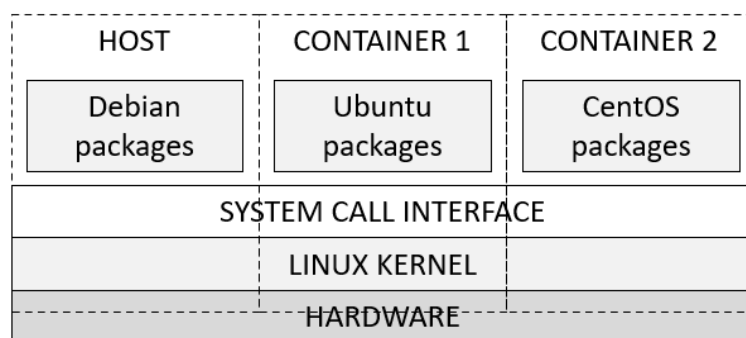
Figure 3: Containerized Linux operating systems on a host.

Now, a container is something that can come with its own Linux operating system packages. It runs isolated from the host operating system but uses the same shared Linux kernel. This is what makes it possible to for example run a Debian system in a container on a CentOS host (Figure 3).

2.2.2 System call interface

In Linux terminology, everything that is executed by the machine is called a task. Tasks are executed in either kernel mode or user mode. The kernel executes tasks in kernel mode, while for example a container living in user space executes tasks in user mode.

In general, kernel mode is reserved only for the lowest-level and most trusted functions of the operating system. When a task is executed in kernel mode, it has complete and unrestricted access to the underlying hardware and can call any CPU instruction or reference any memory address. Tasks executed in user mode have no ability to directly access the hardware or reference memory, but need to delegate to kernel mode to achieve this.

Tasks can be delegated from user mode to kernel mode through a set of program interfaces that are known as system calls. System calls can be thought of as an API that the kernel presents to user mode. They allow requesting the kernel for services such as open file or create process.

System calls can be made in two different ways. The easier way is through libraries such as GNU C (libc) that provide wrapper functions for system calls. A wrapper function accepts system call arguments as function arguments on the stack, initializes the registers using these arguments, and executes the assembler instruction to switch execution to kernel mode. Then, once the kernel has executed the task, it returns control to the user mode, the wrapper function examines the result status, and returns it as its return value [8].

System calls that don't have a wrapper function must be called manually. At the assembler level, a caller assigns the unique system call number and arguments to particular registers and executes a special instruction to switch the processor to kernel mode to execute the system call handling code. Then, upon return, the kernel places the result status into a particular register and executes a special instruction to return the processor back to user mode. The caller can then read the response status from the registers.

As system calls always results in kernel mode execution, it's important that tasks are not overly privileged and have only the necessary capabilities. Therefore, in the context of containers, a host should always restrict the capabilities of a container. Otherwise there would be a risk that a malicious container could use system calls to compromise the entire host.

In Linux, there are two types of processes: privileged (whose effective user ID is zero) and unprivileged (whose effective user ID is nonzero) [9]. Privileged processes, referred to as superuser or root, can bypass all the kernel permission checks. Unprivileged processes are subject to full permission checking based on the process's credentials. Process privileges are divided into distinct units known as capabilities that can be independently enabled or disabled, and are a per-thread attribute.

Linux containers are not "Linux container primitives" but groups of processes as-

sociated with common Linux tools. These tools are used to enable the operating-system-level virtualization that essentially is just processes with certain capabilities. From the kernel's point of view, a container is just like any other process run on the host. The kernel doesn't care if the container process is running a Linux system different from the host, but is only interested in its capabilities.

2.2.3 Docker

Docker is a tool that was designed to make it easy and safe to create and use application containers. It originally started off as an internal project in a company called dotCloud, but was later released as open source in 2013. Since then, it has become the most popular container engine.

In its first release, Docker used LXC as its default execution environment for containers. With LXC, Docker containers behaved like lightweight virtual machines and could be treated like operating systems. They could be logged in and installed multiple applications, and they would work as expected.

In 2014, with the release of Docker version 0.9, LXC was replaced with *libcontainer* [10], and Docker became a project to build single application containers. The Docker base operating system template was trimmed down to a single application environment, and Docker containers were restricted to a single application by design.

Libcontainer [11] was developed to serve the explicit purposes of Docker. It's a layer of abstraction that wraps system calls to the Linux kernel containment features such as namespaces, cgroups, capabilities, and filesystem access controls. It can be called from the Linux user space to manage the life cycle of a Docker container and allows, for example, running an application in a container, performing additional operations on a created container, or destroying a container [12].

Libcontainer later grew into *runc* that became the basis of container runtime specification describing the life cycle of a container and the behavior of a container runtime. It was donated to the Open Container Initiative (OCI) in 2015 followed with a donation of the Docker image format [13]. The OCI is a Linux Foundation project that aims in bringing standardization to container technologies.

While runc is an actual container runtime, another layer of software is usually used to manage it. With Docker, this software is *containerd*. Containerd extends the scope of the container life cycle management over that of runc. It provides functions such as container image download, storage and network interface management, and calls runc with the right parameters to run the containers.

In 2017, containerd was donated to the Cloud Native Computing Foundation (CNCF) becoming an industry-standard container runtime [13]. CNCF is another Linux Foundation project to promote containers. Both runc and containerd can be used with other OCI compatible container runtimes such as *rkt* to run containers.

On top of the Docker component stack sits the Docker engine. It's a daemon process called *dockerd* and serves all the features of Docker through a REST API. It

utilizes containerd to create and manage Docker objects such as images, containers, networks, volumes, and plugins. Docker also provides a user friendly command line interface that communicates with the dockerd REST API.

Some of the core Docker objects [14] are the following:

Images. An image is a template with instructions for creating a Docker container. An image can be based on another image that is based on a third image and so on. Docker images are created as *Dockerfiles* with a simple syntax for defining the steps needed to create the image and run it as a container.

Containers. A container is an instance of an image that can be run. It's defined by both the image and any configuration options provided when created or started. A container can be created, started, stopped, moved, or deleted. It can also be connected to one or more networks, attached storage, or turned into a new image based on its current state.

Services. A service allows to scale containers across multiple computers. Each computer runs an instance of a Docker daemon that then work together. The service is defined a desired state such as the number of replicas of the containerized application that must be available at any given time. The Docker daemons then ensure that the current state of the service matches its desired state.

2.3   Linux container building blocks

A container is typically shipped on a Linux host as a tarball (tape archive file, TAR) that contains something that looks like a Linux file system. The tarball is usually referred to as a container image and it can also include such as built-in applications, host specific configurations, or other files. Container images are what makes it easy to copy and deploy containers on any Linux host.

The contents of a container image are unpacked and run inside a container process on a Linux host. The container process is a normal Linux process but with limited capabilities. The process is also isolated from the host operating system in a way that makes it appear to the Linux system run inside the container as if itself was the host operating system. This way a number of Linux systems can seamlessly coexist on a single host even that they share the same Linux kernel.

The most important Linux kernel features that allow for container process virtualization, isolation, and resource usage controls are namespaces and cgroups. Namespaces are isolation functionalities that allow complete isolation of a process's view of the operating environment [5]. Cgroups, also known as control groups, allow limiting and prioritizing a process's resource usage [4]. In simple terms, namespaces control what a container can see, and cgroups what a container can do.

In the next subsections, Linux namespaces and cgroups are described in detail with practical examples executed in the Linux terminal. Docker is used in the examples to create and run containers. As the primary focus in the examples is in the Linux kernel features, any prior knowledge about Docker is not necessary. Also, in some

examples containers are created with custom scripts written in GO. These scripts can be found from appendices A and B.

2.3.1 UNIX timesharing system namespace

UNIX timesharing system namespace [15] allows creating a separate copy of the hostname and the NIS (Network Information Service) domain name for a process. Hostname is a label that allows identifying a device that is attached to a network. NIS domain name is a client-server directory service protocol for distributing configuration data between computers on a computer network.

In the context of containers, the UNIX timesharing system namespace is useful in that it allows setting each container a unique hostname that is different from the host. This way the different containers are easier to identify and distinguish from one another. As NIS is nowadays rarely used and has been superseded by other protocols, its use is normally less relevant with containers.

The UNIX timesharing system namespace can be observed with Docker:

```
# Create containers
$ docker run -d --name=demo1 debian sleep 60
$ docker run -d --name=demo2 debian sleep 60

# View the containers' hostnames
$ docker exec -it demo1 hostname
> a2bc1768673f

$ docker exec -it demo2 hostname
> f46c90ce56ab

# View the host's hostname
$ hostname
> jviding
```

As can be seen in the example above, both the containers had different hostnames. The hostnames were also different from the hostname of the host. This is possible when a container process is created with its own UNIX timesharing system namespace that allows containers to have their own separate copy of the hostname.

2.3.2 Process ID namespace

In Linux, every process in execution has its own process ID (PID). The process data, including PID, is stored to a special type of a filesystem called proc filesystem (procfs). Procfs provides a method to communicate between kernel and user space and it can be thought of as a file-based interface to process data in the Linux kernel. It's typically mapped to a mount point named /proc.

The /proc mount point can be observed with Docker:

```
# Create a container
$ docker run -d --name=demo debian sleep 60

# View procfs inside the container
$ docker exec -it demo ls /procfs
> 1, 17, ...

# View procfs on the host
$ ls /procfs
> 1, 10, 1073, 11, 11075, 11083, ...
```

As can be seen in the example above, the contents of the procfs are different in the container and on the host. This is because of that the container uses its own procfs shipped with its own Linux system instead of the host's procfs. This prevents the container from accessing the host's process data.

The Linux process model is a single tree-like hierarchy. A process started by another process is called a child process. Therefore, all the processes on a Linux system are children or grandchildren to the init process which is executed when the kernel at boot time starts the system.

A container process running on a host is child to the process that started it. Processes run inside the container are grandchildren to the process that started the container. Therefore, the container and any process run inside it are also linked to the host procfs.

This can be observed with Docker:

```
# View PID of sleep process in the container
$ docker exec -it demo pidof sleep
> 1

# View PID of sleep process on the host
$ pidof sleep
> 9847
```

As can be seen in the example above, the sleep process run inside the container is visible to both the container process and the host operating system. However, they both have a different PID for the sleep process. This is because of the process ID namespace [16].

Process ID namespaces isolate the PID number space, meaning that processes in different process ID namespaces can have the same PID. For example, in the earlier example the sleep process had PID 1 in the container while on the host PID 1 belongs to the init process that started the system. With a process ID namespace a

container doesn't need to be aware of the host and can use its own procfs with any PID it wants.

Process ID namespaces also make it easier to migrate containers on new hosts. A set of processes in a container can be suspended for the duration of the migration. Then, when the container including its processes are resumed on the new host the processes can maintain the same PIDs [17].

2.3.3   Mount namespace

In Linux, filesystems are used to control how data can be stored or provided from a physical memory device in a virtual way. All the files in a filesystem are arranged into a big tree-like structure that is also known as the file hierarchy. The root of this hierarchy is at /.

Mounting a filesystem means attaching it to the file hierarchy. A filesystem can be mounted more than once. This is what allows mounting a filesystem from the host to a container process. All mount behavior is controlled by the Linux kernel.

If a container process was created without a new mount namespace, it would get access to the entire filesystem mounted on the host. The container could then read and modify this filesystem. With containers, this would not be very desirable as a malicious container could negatively impact the host.

With a mount namespace, a container's access to the host's filesystem can be limited [18]. The container can be mounted only the necessary parts of the host's filesystem. This allows the container to have its own filesystem that is isolated from other parts of the host's filesystem.

This can be observed with Docker:

```
# Create a container
$ docker run -d --name=demo debian sleep 60

# View the root of the container's filesystem
$ docker exec -it demo ls /
> bin dev home ...

# Find the container's filesystem from the host
$ docker inspect --format='.GraphDriver.Data.MergedDir' demo
> /var/lib/docker/overlay2/4f9677e...9d1a94b/merged

# View the container's filesystem from the host
$ ls $(docker inspect --format='.GraphDriver.Data.MergedDir' demo)
> bin dev home ...
```

As can be seen in the example above, the container's filesystem is part of the host's filesystem. Within the container the mounted filesystem looks like a normal Linux

filesystem as if the container was the host itself. However, the filesystem mounted to the container is actually only a subset of the host's filesystem.

Filesystems can also be mounted to containers as temporary filesystems (tempfs). Tempfs is a special kind of a filesystem that is stored in volatile memory instead of a persistent storage device. Volatile memory is not only fast but it's also cleared when the system is rebooted. This makes it a good choice for such as sharing secret files with containers as they are not persisted on the host.

2.3.4 Network namespace

In Linux, networking is managed by the network stack. The network stack is controlled by the kernel and can be requested for services through the system call interface. It's effectively what allows processes to access a network through such as physical networking devices.

Network namespace is logically another copy of the network stack with its own routes, firewall rules, and networking devices [19]. It can be used to isolate container networking from the host networking. By default, a process would inherit its network namespace from the parent process.

If a container was created without its own network namespace, it would be able to request services from the network stack of the host. This would allow the container to communicate with other hosts in the local network, with processes run on the same host, or even with hosts in the public Internet. With a network namespace the networking capabilities of a container can be better controlled.

This can be observed with Docker:

```
# Create a container
$ docker run -d --name=demo debian sleep 60

# Show network interfaces available to the container
$ docker exec -it demo ip addr
> 1: lo: ...
> 2: eth0@if25: ... inet 172.17.0.3/16

# Show network interfaces available on the host
$ ip addr
> 1: lo: ...
> 2: eth0: ... inet 10.0.2.15/24
> 3: docker0: ... inet 172.17.0.1/16
```

As can be seen in the example above, both the container and the host have an Ethernet interface named eth0. However, these both have a different IP address. This is because the container is running inside its own network namespace with a network stack that is different from the host network stack.

It can also be seen in the example above, that the IP addresses of both the docker0 interface on the host and the eth0 interface in the container belong to the same IP address space. The docker0 interface is actually a virtual bridge that Docker creates by default and the container is connected to it. This is to allow the container with access to the network through the host's network stack.

2.3.5 Interprocess communication namespace

In Linux, interprocess communication (IPC) is a mechanism provided by the kernel. It allows processes to communicate with one another through means such as pipes, shared memory, message queues, and semaphores. By default, a process inherits the IPC environment of its parent.

If a container had access to the IPC environment of the host, it could communicate with other processes run on the host or tamper with their shared memory. This could potentially put these other processes at risk, as a malicious container could try compromising them. Therefore, containers should be created with an interprocess communication namespace [15] to run them within an isolated IPC environment.

The IPC environment isolation can be observed with Docker:

```
# View the IPC environment on the host
$ ipcs -a
> --- Shared Memory Segments ---
> key            owner
> 0x0052e2c1     postgres

# Create a container
$ docker run -d --name=demo debian sleep 60

# View the IPC environment in the container
$ docker exec -it demo ipcs -a
> --- Shared Memory Segments ---
> none
```

As can be seen in the example above, the container has a different IPC environment from the host. On the host there was for example a shared memory segment with the postgres process. However, because of the interprocess communication namespace the container's IPC environment is isolated from the host and it can't communicate with the host's postgres process.

2.3.6 User namespace

From the kernel's point of view, Linux users are just numeric identifier (ID) values. The actual usernames mapped to these ID values are an operating system feature

that the kernel doesn't care about. Linux has two types of users: privileged and unprivileged. The simple difference between these two is that a privileged user can do something while an unprivileged can't.

For example, view the root user:

```
# Show root user
$ cat /etc/passwd | grep root
> root:x:0:0:root:/root:/bin/bash

# The output translates to
# root:x: <UID> : <GID> :root:/root:/bin/bash
```

In Linux, each user is associated with a user ID (UID) and a group ID (GID). UID is a unique positive integer assigned to each user. The kernel identifies and controls the users based on their UID.

GID values allow grouping users. A group can be delegated additional capabilities in an organized fashion. For example, instead of making an unprivileged user a superuser, the user can be assigned to a group with a specific capability. This allows for more fine-grained tuning of user privileges.

File permissions are a good way to observe how Linux implements user controls. There are three different types of file permissions: read, write, and execute. Each file is also associated with an owner and a group.

File permissions can be observed as follows:

```
# Create and show a file
$ touch FILE && chmod 764 FILE && ls -l FILE
> -rwxrw-r-- 1 root root 0 May 24 11:16 FILE

# This translates to
# Owner permissions: rwx
# Group permissions: rw-
# Other permissions: r--
# Owner: root
# Group: root
# File: FILE

# Wherein
# r = Read
# w = Write
# x = Execute
```

As can be seen in the example above, files can be set with specific permissions for owner, group, and other. The owner permissions define what the owner can do with

the file, group permissions define what users belonging to the group can do with the file, and other permissions define what any user can do with the file. The kernel then uses a user's UID to determine whether the user is privileged for a certain operation with the file.

By default, processes inherit the user rights of their caller. Running a process as the root user should normally be avoided as the process inherits the superuser rights and gains unrestricted access to the entire system. In the other hand, running a process as an unprivileged user results in a failure when the process doesn't have all the required user rights to execute the necessary tasks.

Docker can be used to observe user permissions of a process executed inside a container:

```
# Create a container
$ docker run -d --name=demo debian sleep 123

# View the sleep process on the host
$ ps aux | grep sleep
> root ... sleep 123
```

As can be seen in the example above, the sleep process run inside the container had superuser rights on the host. Now, if the sleep process was malicious and managed to escape from the context of the container, it could easily compromise the entire host. To reduce this risk, the sleep process should be run with lower privileges:

```
# Run sleep as nobody in the container
$ docker exec -itd demo su -s /bin/bash -c 'sleep 321' nobody

# View the sleep process on the host
$ ps aux | grep sleep
> nobody ... sleep 321
```

As can be seen in the example above, the sleep process is now run inside the container as nobody. Nobody is a default Linux user that can be used to execute tasks that don't need any special permissions. Now, if the sleep process was malicious and managed to escape from the context of the container, it could no longer easily compromise the entire host as it would only have limited privileges.

However, in some cases it might be necessary to run processes within a container as superuser. With a user namespace superuser privileges can be granted to the context of the container while the container process itself is unprivileged on the host. User Namespaces allow isolating the security-related identifiers and attributes such as UID and GID from the host environment [20].

Docker doesn't use user namespaces by default, so the script from annex A can be used instead to demonstrate user namespaces:

```
# Run the script as a normal user
$ go run container.go run whoami && sleep 123
> root

# View the sleep process on the host
$ ps aux | grep sleep
> jviding ... sleep 123
```

As can be seen in the example above, the sleep process is now run inside the container as root. However, on the host the same process is shown to be run as a normal user called jviding. This is because the custom container process was created with a user namespace that maps the host user inside the container to a root user by changing both its UID and GID to 0. This way a process can have superuser rights inside a container while the same time it's unprivileged on the host.

2.3.7 Control groups

Processes that have no resource constraints can use as much of a given resource as the host's kernel scheduler allows. With containers this can lead into a situation where one container exhausts the host's resources while the others are left to starve. To avoid this, containers should only be allocated a limited amount of the host's resources.

Control groups (cgroups) are a Linux kernel feature that allow defining collections of processes with resource usage limits. They are arranged into tree-like hierarchies in which resource usage limits can be defined at each level. The defined limits can't be exceeded and a child cgroup inherits the attributes of its parent cgroup.

The cgroup filesystem is mounted at /sys/fs/cgroup and it contains mounted subsystems known as resource controllers. These are kernel components that can be used to limit such as CPU time, system memory, or network bandwidth. Multiple resource controllers can be co-mounted against a single cgroup filesystem to manage the same hierarchy of cgroups.

The mounted cgroup subsystems can be viewed as follows:

```
# View the mounted cgroup subsystems
$ mount | grep cgroup
> cgroup on /sys/fs/cgroup/cpu,cpuacct type cgroup
> cgroup on /sys/fs/cgroup/pids type cgroup
> cgroup on /sys/fs/cgroup/memory type cgroup
> ...
```

In the example above, the cgroup called pids can be used for example to limit the number of processes that a container can create. A new cgroup can be created by creating a new directory to the root of the pids filesystem. The kernel will

automatically populate this directory with a set of files that define general properties for the cgroup.

A new cgroup can be created as follows:

```
# Create a new cgroup under pids
$ cd /sys/fs/cgroup/pids/
$ mkdir example && ls example/
> cgroup.procs pids.current pids.events pids.max ...
```

As can be seen in the example above, the kernel automatically populated the new directory with files. From these files the pids.max and cgroup.procs are the most essential when limiting the number of processes that a container can create. File cgroups.procs contains PIDs of the processes that belong to this cgroup. File pids.max defines the maximum number of processes that these processes can create.

The script from annex B can be used to run a custom container that creates a new cgroup:

```
# Run the script
$ go run container.go run /bin/bash

# View the new cgroup on the host
$ cat /sys/fs/cgroup/pids/container/pids.max
> 20

$ cat /sys/fs/cgroup/pids/container/cgroup.procs
> 26349 26354
```

As can be seen in the example above, the script created a new cgroup called container. The cgroup limits the maximum number of processes that can be created to 20. It also contains a collection of two processes of which the first one is the container process itself. The second process is the /bin/bash executed inside the container because it inherited its cgroup attributes from the container process.

A fork bomb can be used to verify that the cgroup limits the container's capability to create new processes. Fork bomb is a kind of a denial of service attack in which a process continually replicates itself. If the cgroup didn't work, the host would quickly become exhausted.

Run a fork bomb inside the custom container:

```
# Run the custom container process again
$ go run container.go run /bin/bash

# Execute a fork bomb inside the container
$ :() { : | : & }; :
```

```
# Verify on the host that the cgroup works
$ cat /sys/fs/cgroup/pids/container/cgroup.procs
> 26618 26623 26636 26640 26642 26643 26644 ... 26665
```

In the above example, the contents of the file cgroup.procs can be viewed multiple times. It can be noticed that the number of PIDs never exceeded 20. Thus, the cgroup worked as intended and protected the host from the fork bomb attack executed inside the container.

2.4 Networking in Linux

The OSI and TCP/IP models define how systems can be interconnected to enable networking. They define a layered architecture that is also followed in the design of the Linux network stack. Linux network stack is in the core of networking in Linux and it can be best understood by understanding the OSI and TCP/IP models first.

2.4.1 OSI model

The Open Systems Interconnection (OSI) model [21] is a conceptual model for understanding data communication functions between any two networked systems. It promotes the idea of interoperability without regard to the underlying internal structures and technologies. It's a product of the Open Systems Interconnection project at the International Organization for Standardization (ISO).

OSI model partitions communication processes into seven abstraction layers. Each layer provides specific functions to support the layers above it and relies on the functions of the layers below it. The three lowest layers provide the functionality to transmit data between two networked systems. The top four layers complete the process in an end system.

In OSI model, the functionality provided to an N-layer is described by the (N-1)-layer, where N is one of the seven layers of protocols operating in the local host. Each layer implements protocols and functions tailored to a specific type of data exchange. Together, these communication protocols and functions on each layers enable an entity on one networked system to interact with a corresponding entity at the same layer on another networked system.

In data transmission, the N-layer concatenates the payload, called the service data unit (SDU), with protocol specific headers, footers, or both. The concatenated payload is called the protocol data unit (PDU). The PDU is then passed to the (N-1)-layer which processes it as a SDU concatenating it into a (N-1)-layer PDU and passing on to the (N-2)-layer. This process continues until the lowermost layer is reached.

After the lowermost layer has produced PDU, its transmitted to the receiving system. The receiving system processes the PDU as a series of SDUs from the lowest

to the highest layer. Each layer strips the PDU from the layer's header or footer and passes the SDU to the above layer. The process continues until the topmost layer is reached where the payload is then finally consumed [22].

The seven layers of the OSI model (Figure 4) are the following:
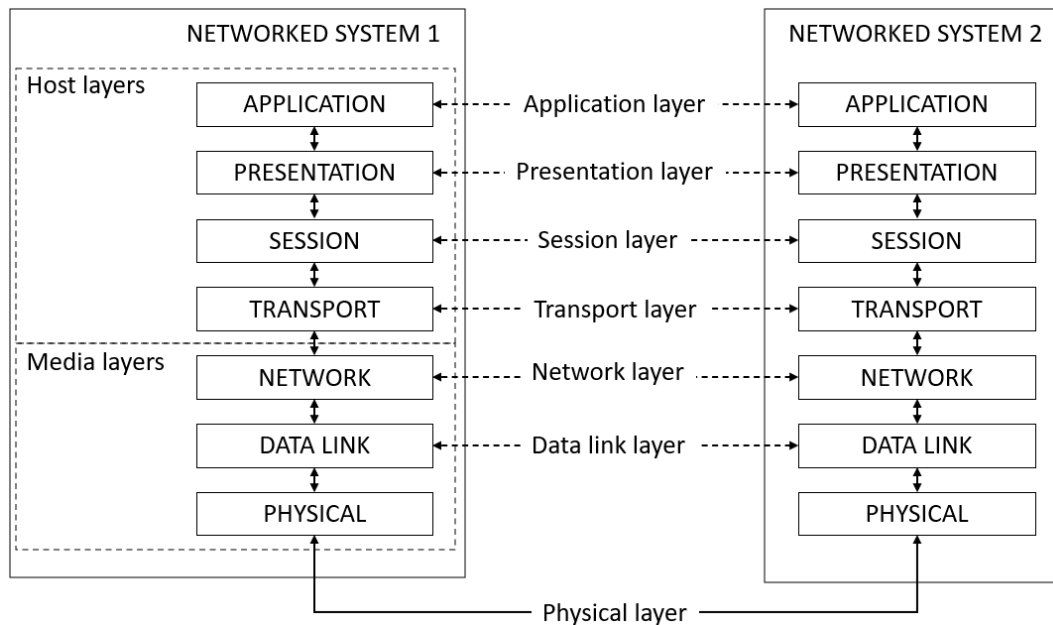

Figure 4: The seven layers of the OSI model. The three lowest layers provide the functionality to transmit data between two networked systems. The top four layers complete the process in an end system.

Layer 7 - Application. The topmost layer in the OSI model is called the application layer. It's what the end user sees and interacts with. It displays the user with data received by applications that implement a networking component. Such application can be for example a web browser or e-mail.

Layer 6 - Presentation. Presentation layer serves as the data translator for the network. It ensures that data sent out by the application layer of another system is readable and understandable to the application on the same system and vice versa. For example, EBCDIC-coded text file may be converted to an ASCII-coded text file during transmission. Presentation layer is sometimes also called the syntax layer.

Layer 5 - Session. Session layer controls the connections between applications that function on different networked systems. It allows the applications to establish, manage, and terminate dialog through the network. For example, a session acknowledges the number of bytes received to the other end of the session.

Layer 4 - Transport. Transport layer is responsible for delivering data to the appropriate application processes on networked systems. The most common transport layer protocols are the Transmission Control Protocol (TCP) and the User Datagram Protocol (UDP). TCP transfers the data in segments and UDP in datagrams. Applications are typically addressed with source and destination port numbers that

would then be included in the headers of each transport layer PDU.

Layer 3 - Network. Network layer provides the means of transferring data between systems across one or more networks. The data is transferred in packets and systems are typically addressed with IP addresses. The source and destination IP addresses would then be included in the headers of each data packet. To reach systems in different networks, packets can be forwarded from one network to another by intermediate routers.

Layer 2 - Data link. Data link layer provides the means of transferring data between two directly connected systems in a network. The data is transferred in frames and systems are typically addressed with MAC addresses. The source and destination MAC addresses would then be included in the headers of the data frame. For example, most switches operate at the data link layer.

Layer 1 - Physical. The lowermost layer in the OSI model is called the physical layer. It's responsible for the transmission and reception of raw bit streams on a networked system. For example, digital bits can be converted into optical signals that are then sent in an Ethernet cable.

2.4.2 TCP/IP protocol suite

TCP/IP protocol suite [23] is named after its most important protocols: Transmission Control Protocol (TCP) and Internet Protocol (IP). It's official, but less used, name is the Internet protocol suite. It was originally developed by the Department of Defence in the United States.

Unlike OSI model, that is just a conceptual framework for understanding application communications over a network, TCP/IP is a model for building interconnected networks that provide universal communication services. It specifies and standardizes communication mechanisms such as interfaces, protocols, and services and that how data should be packetized, addressed, transmitted, routed, and received.

TCP/IP organizes networking operations into four layers of abstraction. This layered representation in known as the protocol stack. Each layer in the protocol stack classifies its related protocols, and provides services to those above it and makes use of the services provided by those directly below it via concise interfaces. This makes development, testing, and implementation of new networking components easier and independent of the underlying structures and technologies.[24]

The four layers of the TCP/IP protocol suite (Figure 5) are the following:

Application layer. Application layer is used by applications for exchanging application data over the network. Such applications are for example web browsers, e-mail services, and Telnet. The interface between the application and transport layer is defined by port numbers and sockets. In OSI model, this layer could be described as a combination of application, presentation, and session layer.

Transport layer. Transport layer provides end-to-end data transfer by delivering data from an application to its remote peer. The transmission is independent of

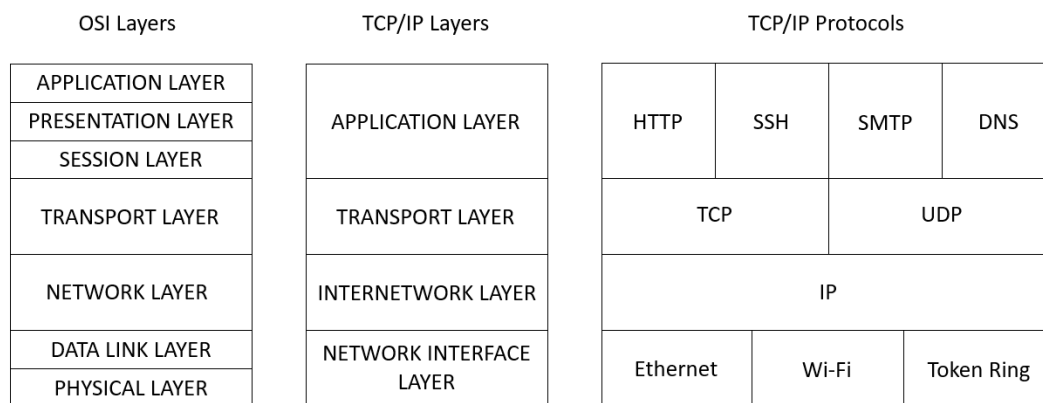| OSI Layers	TCP/IP Layers	TCP/IP Protocols			
APPLICATION LAYER	APPLICATION LAYER	HTTP	SSH	SMTP	DNS
PRESENTATION LAYER					
SESSION LAYER					
TRANSPORT LAYER	TRANSPORT LAYER	TCP		UDP	
NETWORK LAYER	INTERNETWORK LAYER	IP			
DATA LINK LAYER	NETWORK INTERFACE LAYER	Ethernet	Wi-Fi	Token Ring	
PHYSICAL LAYER					

Figure 5: The four layers of the TCP/IP protocol suite with some of their standardized protocols. These layers can also be loosely mapped to the seven layers of the OSI model.

both the application data and the underlying network. The protocols defined by this layer are TCP and UDP. In OSI model, this layer would correspond to the layer with the same name: transport layer.

Internetwork layer. Internetwork layer, also known as internet layer or network layer, is responsible for delivering packets across one or more networks. The most important protocol of this layer is the Internet Protocol (IP) that provides a routing function to deliver transmitted messages to their destination. The other protocols of this layer are ICMP, IGMP, ARP, and RARP. In OSI model, this layer would correspond to the network layer.

Network interface layer. Network interface layer, also known as link layer or data-link layer, is the interface to the actual network hardware. It doesn't describe or standardize any network-layer protocols but standardizes an interface for the internetwork layer to access these protocols. It's responsibility is to move packets between the internetwork layer interfaces of two different systems on the same link. This can be done for example over a physical medium such as an Ethernet cable. In OSI model, this layer would correspond to the data link and physical layer.

2.4.3   Linux network stack

The Linux network stack is what allows applications run on a Linux operating system to access a network through a physical networking device. These devices can be such as WiFi devices, Ethernet cards, or Token Ring cards. The implementation of the Linux network stack is based on the industry standard Berkeley socket API, which has its origins in the BSD Unix development [25].

The Linux network stack is organized into seven layers [26] with clean sets of interfaces. The organization is comparable to the OSI and TCP/IP models. Each layer has a specific job to perform and together they allow the Linux system to communicate over a network.

As the Linux network stack can both send and receive data, the jobs at each layer are performed twofold. Requests start from the upmost layer, the application layer, and go down through the various layers until the lowermost layer, the physical hardware. The responses received from a network then enter the lowermost layer and go up through the various layers until the upmost layer is reached [27].

The layers exist in Linux user space, kernel space, and physical layer. The layers are grouped as follows:

1. The topmost layer resides in user space and it's called the application layer.

2. The next five layers reside in kernel space and they are called the system call interface, protocol agnostic interface, network protocols, device agnostic interface, and device drivers.

3. The final layer resides in physical layer and it's called the physical device hardware.

Next, the seven layers of the Linux network stack (Figure 6) are described with more detail:

Figure 6: The seven layers of the Linux network stack.
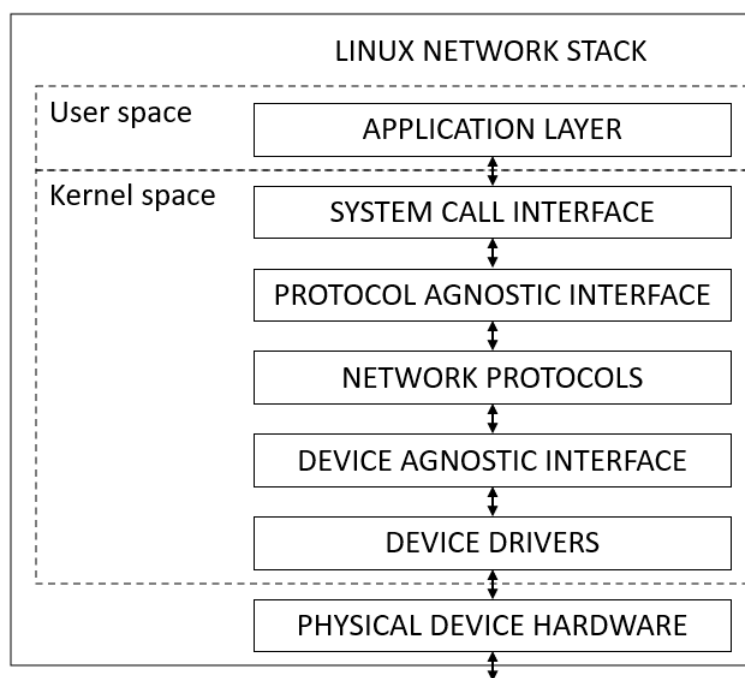
Application layer. Application layer is where all the user applications are. They use an interface called socket API that wraps different system calls to the protocol agnostic interface. The protocol agnostic interface is part of the Linux kernel networking subsystem and defines a structure called socket that the applications use for all the network programming functions.

Socket is a logical endpoint for network communications and works similar to files. Writes to a socket are turned into packets that are sent out, and packets received from the network can be read from a socket. Each socket has its own unique ID that applications use to access it. However, sockets are better known for their TCP/IP port numbers that allow systems to pass received packets to the right application through the right socket.

System call interface. System call interface allows calls from the application layer to the kernel. It provides the means to transfer control between kernel and user mode execution.

Protocol agnostic interface. Protocol agnostic interface, also known as the sockets interface, is where all communications take place through the Linux network stack. It's the interface between the transport layer protocols in the TCP/IP protocol suite and the protocols above. Applications can call it to open sockets not only in TCP and UDP protocols but also in IP, raw Ethernet, and other transport protocols defined in the Linux system's network protocols [28].

Network protocols. Network protocols define the particular networking protocols that are available in a Linux system. The protocol modules are initialized, registered, and added to a list of active protocols when the system starts. When an application creates a socket with a specific protocol module, the module controls how data is sent or received in the network.

Device agnostic interface. Device agnostic interface connects the network protocols to the hardware device drivers. It provides a common set of functions allowing the lower level network device drivers to operate with the higher level protocol stack. For example, network protocols can call the hardware device drivers through the device agnostic interface to pass them data that needs to be sent out to the network. Also, the device drivers can pass data received from the network to the network protocols by calling the device agnostic interface.

Device drivers. Device drivers manage the physical network devices. Such is for example an Ethernet driver over an Ethernet device. They prepare the data received from upper layers to be transmitted from the physical network device to the network over the medium. Also, they read the data received from the network and pass it to the higher layers.

Physical device hardware. Physical device hardware is the actual network hardware. This is where the data packets are transmitted and received over the network medium, whether cable or wireless.

2.4.4 Socket buffers

In Linux, each socket has its own buffer space where data can be stored [29]. The buffer space is a control structure called sk_buff with a block of memory attached. It's defined in the kernel and is the core of the kernel's network subsystem.

Sk_buff provides the network protocols with consistent and efficient buffer handling

methods that include such as adding protocol headers down the stack, removing protocol headers up the stack, concatenation or separation of data, and convenient access to the header fields[30]. For example, any packet sent by an application or received by the physical device hardware is put into a sk_buff. The network stack then uses the sk_buff to pass the packet through the different layers. This also minimizes the copying overhead.

2.5   Linux network devices

In this section, different Linux network tools and devices are described. These are essential to understand in later sections of this paper where these devices are used to enable networking for containers. The described tools are routing table, forwarding table, ARP cache, network bridge, veth, VXLAN, network interface, and iptables. Overall, these are just a fraction of all those network tools built in the networking subsystem of the Linux kernel.

2.5.1   Routing table

Routing tables[31] provide important functionalities to many of the Linux network devices and also to the kernel. It's a data table that contains information about the topology of the network immediately around it. This information is basically a list of routes to particular network destinations.

When a packet needs to be sent across multiple networks to a remote destination, routing table tells the next destination of the packet where it should be forwarded to. Then the next device, such as a router, that receives the packet, has its own routing table again that tells the next destination the packet should be forwarded to. With this type of hop-by-hop routing packets can be delivered anywhere across multiple networks. In OSI model, routing takes place on the network layer and is usually based on IP addresses.

In Linux, routing tables are populated automatically by the kernel when new network interfaces are brought up with IP addresses. The table entries contain information such as network destination, netmask, gateway, and interface. Network destination combined with the netmask tells the range of IP addresses that can be reached through the interface and gateway.

For example, assume a routing table entry mapped to an Ethernet interface eth0 with network destination 192.168.0.0, netmask 255.255.0.0, and gateway 192.168.0.1. The combination of the network destination and netmask gives a range of IP addresses that is 192.168.0.0 - 192.168.255.255. These are the IP addresses that can be reached through the Ethernet interface eth0 and gateway.

Now, if a packet was to be sent out to a remote IP address 192.168.12.34 it would match the routing table entry as the IP address is in the range of IP addresses calculated from the network destination and netmask. The entry tells that the packet

should be forwarded to the gateway 192.168.0.1 through the Ethernet interface eth0. Once the packet is received by the gateway, the gateway then uses its own routing table to forward the packet to the next network.

2.5.2   Forwarding table

Forwarding tables[32] provide important functionalities for network bridges in Linux. They maintain mappings of MAC addresses with port numbers. In OSI model, forwarding tables operate on the data link layer.

Network devices are connected to bridges through ports. When the bridge receives a packet through port P with destination MAC address M, it adds a new entry to the forwarding table that maps port P with the MAC address M. Now, if the bridge later needed to forward a packet to MAC address M, it could check the forwarding table to learn that the packet should be forwarded through port P.

2.5.3   ARP cache

ARP caches[33] provide important functionalities to many of the Linux network devices and also to the kernel. They maintain mappings of IP addresses with MAC addresses. In OSI model, MAC addresses are used on the data link layer while IP addresses on the network layer.

When a system needs to send an IP packet to a given IP address, it checks its ARP cache to find the corresponding MAC address. If there's no mapping in the ARP cache, the system makes an ARP request with the given IP address to find the MAC address. The result of the ARP request is then stored to the ARP cache. This way the system won't need to make a new ARP request for every IP packet sent.

2.5.4   Network bridge

Network bridge[32] is a type of a network device that connects different systems together and provides communications between them. It forwards packets between the connected systems based on MAC addresses. In OSI model, network bridges work on the data link layer. For this reason they are sometimes also referred to as layer two devices.

Both physical and virtual systems can be connected to a network bridge. Each system is connected to a different port on the bridge. The bridge works transparently and the systems connected to it don't need to know or care about its existence. The bridge maintains a forwarding table that it uses to make packet forwarding decisions including where to pass, transmit, or discard a packet[34].

2.5.5 Veth

Veth devices[35] are virtual Ethernet devices that are always created in interconnected pairs. They could be thought of as Ethernet cables that can be used to combine the network facilities of the Linux kernel together in interesting ways. For example, veth pairs can be used to tunnel network traffic between network namespaces. A packet transmitted on one device is immediately received by the device in the other end of the pair. If either veth device is down, the link state of the pair is down.

2.5.6 VXLAN

VXLAN is a tunneling protocol for running an overlay network on existing IP networks[36]. It could be described as Ethernet in a UDP tunnel. VXLAN functionality has been implemented with the kernel tunnel device since Linux 3.12[37] and it's described in IETF RFC 7348[38].

VXLAN allows connecting devices in different networks together as if they were all part of the same network. The connections can be thought of as virtual or logical links that are transparent to the devices, but correspond to paths that may go through many physical links in the underlying network. Thus VXLAN overlay networks can be used to decouple network services from the underlying network infrastructure.

The control plane of a VXLAN overlay network is formed with VXLAN tunnel endpoints (VTEP)[39]. They provide the functionality to deliver Ethernet frames in UDP tunnels across different networks. In the delivery, each frame is added a VXLAN header before encapsulated in a UDP packet.

Each VXLAN header contains a VXLAN network identifier (VNI) that allows the receiving VTEP to forward the Ethernet frame to the correct network. The maximum size of the VNI is 24 bits (16777216). This is also designed to solve the problem of limited VLAN IDs (4096) in IEEE 802.1q[40]. However, this comes with a cost. VXLAN headers add a 50-byte-overhead to every frame the VTEPs deliver[41].

Network links of type VXLAN maintain MAC and IP address mappings of destinations on remote networks. They can also use these mappings to answer ARP queries in their own network and so work as ARP proxies. By default, the mappings are empty, but can be learned in several ways. A common practice is to just create them manually, as is the case with for example Docker[42].

2.5.7 Network interface

Network interfaces[43] are interfaces to network devices, whether virtual or physical, and configure how the Linux network stack is connected to a network. All packets sent or received by the Linux network stack flow through the network interfaces. One of the most typical network interfaces in a Linux system is called eth0 and it

interfaces an Ethernet device.

2.5.8   Iptables

Iptables [44] is a flexible firewall utility built in to the Linux system. It uses policy chains with lists of rules for what to do with a packet. In OSI model, iptables work primarily on the network and transport layer.

The policy chain types include such as input, forward, and output chains. Input chains control incoming connections. Forward chains control incoming connections that aren't destined to the system itself. These are typically used only if the system needs routing capabilities. Output chains control outgoing connections.

Each chain contains a list of rules that define what to do with a packet. Iptables matches packets that arrive or leave the system against these rules one by one. If a packet does not match, the next rule in the chain is examined. It a packet does match, the action defined in the rule is taken. A rule can define a jump to another policy chain or one of the values accept, drop, queue, or return. If none of the rules match, the default policy defined in the chain determines the fate of the packet.

Accept rule lets the packet through to its destination. Drop rule drops the packet meaning that the connection is rejected. Queue rule passes the packet to user space where it can be received by an application. Return rule instructs the iptables to stop traversing this chain and resuming at the next rule in the previous chain. If there's no previous chain to resume, the default policy defined in the chain determines the fate of the packet.

Iptables contains also a number of other features. For example, with connection tracking it can be controlled that only the other end of communications can open a new connection while the other end can only communicate if there's an already established connection. Iptables supports also features such as NAT that can be used to mask the outgoing or incoming packets with another IP address. Together these features are what makes iptables a very flexible firewall utility.

2.6   Journey of a network packet

To better understand the packet flow through the Linux network stack, assume a user is running a web browser. Web browser is a user space application that works on the application layer. It implements a networking component that uses the socket API to access different types of remote data over the network.

When the user browses to a new website, the browser needs to fetch an HTML file from the remote web server. Its networking component uses the socket API to make a system call to the protocol agnostic interface to open a new socket for a TCP connection. The protocol agnostic interface then requests the TCP module from the network protocols and creates the socket. The new socket is assigned an ID that will identify it with the web browser application. The socket is also assigned a

TCP/IP port number that identifies it within the system.

Once the socket is ready, the browser writes it the destination IP address and port number. It also writes the HTTP request data to the socket. The data is stored to the sk_buff structure.

Write to the socket activates the TCP module. The module builds the TCP headers to the sk_buff structure that now represents a TCP segment. The added headers contain information such as the source and destination IP addresses and port numbers. Next, the TCP module calls the IP module in the network protocols to pass it control[45].

The IP module performs the routing operations for the TCP segment[46]. It checks the routing table for which network device provides the best route for the packets. If a route doesn't exist, the packet is dropped. If a route exists, the IP module writes the source and destination IP and MAC addresses to the sk_buff structure that now represents an Ethernet frame. The destination MAC address is resolved from the ARP cache or with an ARP request[47].

Once the packet is ready for transmit, the IP module makes a call to the device agnostic interface. The packet described by the sk_buff structure is now moved to the transmission queue of the device driver. The device driver eventually sends the packet out through the physical device hardware to the network where it will be routed to its destination[48].

When the packet is received by the destination host, the handling is inverse, but with some differences. The packet data is received by the device driver through the physical device hardware. The device driver converts the data into a sk_buff structure and then adds it into a backlog queue.

When network protocols are initialized in a Linux system, for example upon a system start, each protocol registers itself with a receive data processing routine. This routine allows the device driver to make an indirect call to a protocol and pass it control of a packet. In this case, the device driver calls the IP module to pass it control.

The IP module parses the headers of the packet, verifies their validity, strips them from the sk_buff structure, and calls the TCP module to pass it control. The TCP module performs the TCP protocol specific processing, checks the destination port of the packet to determine its destination socket, strips the sk_buff structure from the remaining headers, and passes control to the server application listening on the socket. The server application reads the HTTP request data from the socket, processes it, and sends a response.

3   Container networking

One of the reasons containers are so powerful is that they can be connected to other containers and non-container workloads in a number of different ways. For example,

with Docker containers can be set up networking in modes that are called none, host, macvlan, bridge, and overlay [49]. There are also a number of different networking plugins that allow for other customized networking modes.

In none mode networking, all networking is disabled for the container. This simply means that the container has no network access. This mode is usually used in conjunction with a custom network driver to set up some other customized networking mode.

In host mode networking, the container uses the host's network namespace directly. This means that the container uses the network stack of the host and there's no networking isolation between them. The container has the IP address of the host and performs networking as any application run directly on the host. This type of networking mode is useful when only other parts of the container need to be isolated.

In macvlan mode networking, the container appears in the network as a different physical network device. Instead of routing the packets through the host's network stack, the container is assigned a MAC address and it's directly connected to the physical network interface of the host. As this may result in networking issues due to IP address exhaustion or to VLAN spread, which is a situation in which there are too many MAC addresses in a network, bridge or overlay mode networking is usually preferred. Macvlan mode networking may be useful if migrating from a virtual machine setup or if there are legacy applications that expect to be directly connected to the physical network.

In bridge mode networking, the containers are interconnected through a virtual bridge device. Bridge networks apply to containers running only on the same host. To enable communications between containers running on different hosts, the routing can be managed either at the operating system level or with overlay mode networking. With Docker, bridge mode networking is the default.

In overlay mode networking, isolated bridge mode networks are interconnected through VXLAN tunnels. This allows the containers to communicate across different bridge mode networks without the need for operating system level routing. Also, as the container communications are tunneled the communications are agnostic to the underlying physical network.

In this section, bridge and overlay mode networking for containers are described in detail. The descriptions are supported with practical examples executed in the Linux terminal. Also, the VXLAN control plane is described in detail and the traffic control subsystem in the Linux kernel. The traffic control subsystem allows limiting the networking capabilities of containers to prevent them from exhausting the host system.

3.1 Bridge mode networking on a single host

In this section, a step by step implementation of a bridge mode container network is presented. Network communications are enabled for two containers by attach-

ing them to this network. The containers are created and run with Docker. The containers are run in none mode networking so that networking can be enabled manually.

The presented bridge mode container network differs from the Docker default bridge mode network in that the bridge is created within its own network namespace. This isolates the network and improves security by ensuring that only the attached containers can communicate within it. Otherwise there would be a risk that also other unrelated services, containers, or similar could reach this network. With Docker, the default bridge is not isolated and its created in the host's network namespace [50].

As this section focuses only on the networking aspects of containers, it's not relevant whether the containers are created with Docker. Any container runtime can be used as long as the containers are created without any networking enabled. The networking is then set up manually using the Linux kernel built-in networking tools.

The first component needed for setting up the bridge mode container network, is a new network namespace in which the bridge can be created. By default, each new network namespace comes only with a loopback interface. This interface is not connected to any physical network device but allows connections to services that are run on the same host machine (localhost) [51].

A new network namespace can be created with the following command:

```
# Create a new network namespace called demonet
$ ip netns add demonet
```

To be able to attach containers to this network, the next required component is a virtual network bridge. It works on the data link layer of the OSI model and maintains a forwarding table to ascertain where to pass, transmit, or discard a data frame [52]. In this case, the network bridge is what will allow the communications for the attached containers by routing packets between them [53].

A new network bridge can be created and set up with the following commands:

```
# Create a bridge with name br0 in demonet
$ ip netns exec demonet ip link add name br0 type bridge

# Set the bridge an IP address
$ ip netns exec demonet ip addr add dev br0 192.168.0.1/24

# Bring the bridge interface up
$ ip netns exec demonet ip link set br0 up
```

Now that there is a network namespace with a bridge, containers can be created and attached to it. Docker will be used to create these containers. Each container
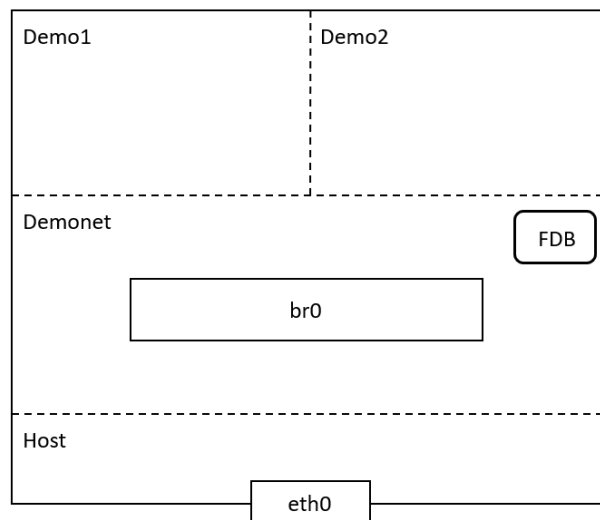
Figure 7: Three isolated network namespaces without any connectivity. Two for containers demo1 and demo2, and demonet with a network bridge br0. The bridge maintains a forwarding table, FDB, that is used for packet routing.

comes with its own network namespace that allows it to work in isolation without facing resource sharing conflicts with other containers running on the same system.

To be able to set up networking for the containers manually, they should be created with the networking mode none [54]. This ensures that the containers are created with all networking disabled and that they are not connected to anything. The newly created containers will now each have its own isolated network namespace in which there is only the default loopback interface.

Docker containers can be created with the following commands:

```
# Create two Docker containers
# The arguments ''sleep 3600'' will keep them alive for an hour
$ docker run -d --net=none --name=demo1 debian sleep 3600
$ docker run -d --net=none --name=demo2 debian sleep 3600
```

There are now three different network namespaces on the host system (Figure 7). One of them is demonet that contains the network bridge. The other two belong to the created containers. To enable networking for the containers, they both need to be attached to the network bridge in demonet. This can be done with veth devices.

Veth devices allow tunneling traffic between network namespaces [35]. They are created in interconnected pairs, and a packet transmitted on one device in the pair is immediately received on the other device - even if the other device was in a different network namespace. To be able to attach the containers to the bridge in demonet, the veth pairs need to be first created and then moved to the right network namespaces.

Veth devices can be created and set up with the following commands:

```
# Create a veth pair for both containers
$ ip link add dev veth1 mtu 1450 type veth peer name veth2 mtu 1450
$ ip link add dev veth3 mtu 1450 type veth peer name veth4 mtu 1450

# Set the first device from both pairs to demonet
$ ip link set dev veth1 netns demonet
$ ip link set dev veth3 netns demonet

# Bring the veth interfaces up
$ ip netns exec demonet ip link set veth1 up
$ ip netns exec demonet ip link set veth3 up

# Get the network namespace path of both the containers
$ ctn_ns_path1=$(docker inspect \
$> --format=''{{ .NetworkSettings.SandboxKey}}'' demo1)
$ ctn_ns_path2=$(docker inspect \
$> --format=''{{ .NetworkSettings.SandboxKey}}'' demo2)

# Get the network namespace name of both the containers
$ ctn_ns1=${ctn_ns_path1##*/}
$ ctn_ns2=${ctn_ns_path2##*/}

# A symbolic link in /var/run/netns is required
# to use the native ip netns commands
$ ln -sf $ctn_ns_path1 /var/run/netns/$ctn_ns1
$ ln -sf $ctn_ns_path2 /var/run/netns/$ctn_ns2

# Set the latter device from both veth pairs to
# the network namespaces of the containers
$ ip link set dev veth2 netns $ctn_ns1
$ ip link set dev veth4 netns $ctn_ns2
```

Veth pairs could be thought of as virtual Ethernet cables in that both the ends need to be plugged into something before a network connection can be established. The created veth devices are now in the right network namespaces but they are not plugged into anything. To start enabling networking, the other ends of the veth pairs need to be plugged into the network bridge in demonet.

The veth devices can be plugged to the network bridge with the following commands:

```
# Plug the veth devices into the network bridge br0 in demonet
$ ip netns exec demonet ip link set veth1 master br0
$ ip netns exec demonet ip link set veth3 master br0
```

Now that the veth pairs are plugged into the network bridge in demonet, the next step is to plug their other ends into something in the container network names-

paces. This can be achieved with Ethernet interfaces. An Ethernet interface allows configuring how a network stack is connected to a network.

When creating an Ethernet interface it should be set MAC and IP addresses so that it's addressable. These will effectively be the MAC and IP addresses of the container running in the network namespace. The IP address can be selected from the IP address range of the network bridge (192.168.0.0/24). With Docker, the MAC address is chosen so that the first 4 bytes are typically 02:42 while the rest of the MAC address is derived from the container's IP address. If the first container is set the IP address 192.168.0.2, the corresponding MAC address with the Docker convention would be 02:42:c0:a8:00:02.

The Ethernet interfaces can be set up with the following commands:

```
# In the container network namespaces:
# 1) Plug veth device as an Ethernet interface eth0
# 2) Set the interface a MAC address
$ ip netns exec $ctn_ns1 ip link set dev veth2 name eth0 \
$> address 02:42:c0:a8:00:02
$ ip netns exec $ctn_ns2 ip link set dev veth4 name eth0 \
$> address 02:42:c0:a8:00:03

# Set the eth0 interfaces IP addresses
$ ip netns exec $ctn_ns1 ip addr add dev eth0 192.168.0.2/24
$ ip netns exec $ctn_ns2 ip addr add dev eth0 192.168.0.3/24

# Bring the eth0 interfaces up
$ ip netns exec $ctn_ns1 ip link set dev eth0 up
$ ip netns exec $ctn_ns2 ip link set dev eth0 up
```

The bridge mode container network is now successfully set up and the containers can communicate with one another over the network bridge using the configured MAC and IP addresses (Figure 8). This can be verified by sending a ping request from one container to the other. However, as this network is isolated, the containers are not able to connect to any external resources.

Ping requests sent:

```
# Send ping requests between the containers
$ docker exec demo1 ping 192.168.0.3
$ docker exec demo2 ping 192.168.0.2

# Ping works
> 64 bytes from 192.168.0.3

# Try to ping a target in the public Internet
# The IP address 8.8.8.8 points to a Google nameserver
```
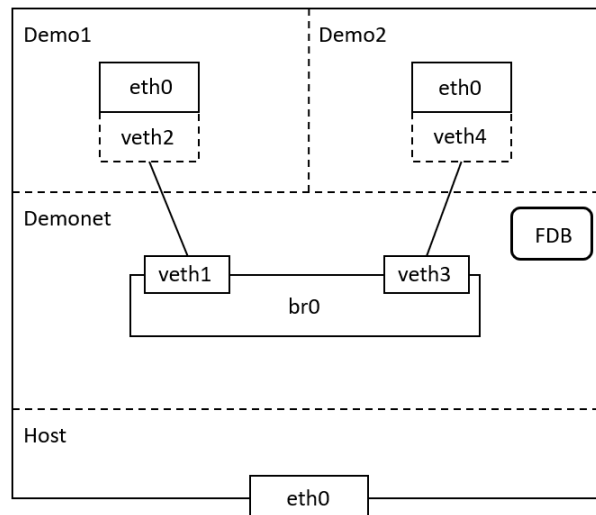
Figure 8: A bridge mode container network. The containers are plugged into the network bridge with veth pairs that work as virtual Ethernet devices.

```
$ docker exec demo1 ping 8.8.8.8
$ docker exec demo2 ping 8.8.8.8

# Ping fails
> 5 packets transmitted, 0 received
```

To allow connections from the created bridge mode network to other networks, a new network path should be created. If connections were needed only to another bridge mode network, a good choice would be to use an overlay mode network. For other connections, operating system level routing is an option.

3.1.1 Operating system level routing

A bridge mode container network that is not connected to any other networks can only route packets between the containers that are attached to it. If none of these containers are attached to any other network, there are no possible paths for packets to any other network. This results in that containers can only communicate with those that are attached to the same bridge mode network.

To be able to reach for example the public Internet, the containers need a network path to it. If the host is connected to the public Internet, this path can be established through the host's network namespace. The first requirement is to create a connection between the host's network namespace and the bridge mode network. This can be done with veth pairs, just as was done with the containers.

```
# Create a new veth pair
$ ip link add dev veth5 type veth peer name veth6
```

```
# Attach the other end to the bridge in demonet
$ ip link set dev veth6 netns demonet
$ ip netns exec demonet ip link set veth6 master br0
$ ip netns exec demonet ip link set veth6 up

# Attach the other end to the host's network namespace
# as an Ethernet interface
$ ip link set dev veth5 name eth1 address 02:42:c0:a8:00:04
$ ip addr add dev eth1 192.168.0.4/24
$ ip link set dev eth1 up

# Verify connectivity with ping
$ docker exec demo1 ping 192.168.0.4
> 64 bytes from 192.168.0.4
```

The bridge mode network is now connected to the host network namespace and the containers can send packets to the host. However, as the containers should also be able to reach the public Internet and not only the host, the host needs to be configured to act as a router. This can be achieved by enabling IP forwarding.

```
# Enable IP forwarding on the host
$ sysctl net.ipv4.ip_forward=1
```

IP forwarding [55] is a process that allows transferring packets between network interfaces. If the interfaces are connected to different networks, IP forwarding can be used to pass packets between them. So in this case, the containers should be able to reach the public Internet by sending packets from the bridge mode network to the host, and the host should then forward these packets to the public Internet.

The containers must also be made aware of that the host can forward their packets to other networks. This can be achieved by adding a new entry to their routing tables. The entry shall tell the containers that destinations that are not within the bridge mode network can be reached through the host. This is also known as the default gateway.

```
# Set default gateways for the containers
$ ip netns exec $ctn_ns1 ip route add default via 192.168.0.4
$ ip netns exec $ctn_ns2 ip route add default via 192.168.0.4
```

Now, to send a packet to the public Internet, the containers can read the entry from the routing table and send the packet to the host for forwarding. As the containers and the host are connected through a network bridge, that operates at the data link layer in the OSI model, the packets are addressed to the host using its MAC address while the destination IP address is that which points to the public Internet.

When the host receives the packet, it reads the destination IP address and forwards the packet to the next device in another network. Eventually the packet is then forwarded to its final destination.

However, the host still doesn't know how to forward the packets received from containers. This can be fixed with an iptables rule. The rule shall tell the host that packets received from the container network through interface eth1 shall be forwarded through the interface eth0 to the public Internet.

```
# Allow forwarding from eth1 to eth0
$ iptables -A FORWARD -i eth1 -o eth0 -j ACCEPT
```

The containers can now send packets to the public Internet. However, the host forwards the packets with the source IP addresses of the containers. These IP addresses are private and only known within the host. If a response is sent back from the public Internet, the intermediate network routers don't know how to route this response.

To be able to receive the responses, the packets should be forwarded with a source IP address that is known to the outside world. This can be achieved by using source network address translation (SNAT) [56]. With SNAT the source IP address of each outgoing packet can be changed to the IP address of the host.

SNAT also stores an entry about each source IP address translation to a connection tracking table. The connection tracking table contains a list of all currently tracked connections through the system. When a response is eventually received, the connection tracking table is searched for a translation. In this case, the translation will tell the host to change the destination IP of the received packet to the original source IP address of the container that is now waiting for the response.

```
# Change source IP of forwarded packets
# to the IP address of the host
$ iptables -t nat -A POSTROUTING -o eth0 \
$> -s 192.168.0.0/24 -j MASQUERADE
```

The host can now forward packets to the public Internet with SNAT and receive the responses. However, it still needs an iptables rule that allows it to forward the responses back to the bridge mode network. For security reasons, this rule needs to be set up appropriately so that the bridge mode network is not accidentally exposed to the public Internet.

As the host should only forward responses to the container network, but not new connections, it needs to track for connections to tell the legitimate response packets from other packets. Connection tracking can be enabled with the iptables conntrack module combined with ctstate. These allow setting the iptables rule so that only packets associated with a legitimate connection are forwarded.
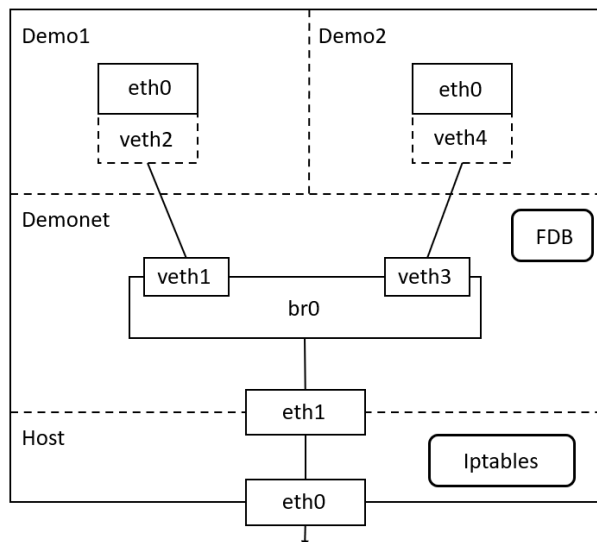
Figure 9: IP forwarding from bridge mode network through host to public Internet with Iptables.

```
# Forward only response packets from eth0 to eth1
$ iptables -A FORWARD -i eth0 -o eth1 -m conntrack \
$> --ctstate RELATED,ESTABLISHED -j ACCEPT
```

Now with the iptables rules set, the containers can communicate to the public Internet through the host's network namespace (Figure 9). This can be verified with ping requests.

```
# Ping from containers demo1 and demo2
$ docker exec demo1 ping 8.8.8.8
$ docker exec demo2 ping 8.8.8.8

# Both pings work as expected
> 64 bytes from 8.8.8.8
```

Linux networking tools are powerful and allow for a variety of different setups for operating system level routing. This section scratched only the surface of all the possibilities that they can provide. For example, Docker uses these tools for additional features such as load balancing traffic between containers.

3.2   Overlay mode networking on multiple hosts

In this section, a step by step implementation of an overlay mode container network is presented. Network communications are enabled between containers that are in different bridge mode networks. The bridge mode networks are set up on different hosts and they are then connected with VXLAN.

VXLAN is a common approach for implementing overlay mode networks with containers. For example, it's used by the Docker overlay network driver that has come built-in with the native Docker since version 1.9 [57]. It's also used by the popular third-party container networking plugins Weave [58], Flannel [59], and Calico [60].

To demonstrate that VXLAN is agnostic to the underlying physical network, the overlay mode container network is first set up with Ethernet connections and then with WiFi connections. The creation of a bridge mode container network was described in the previous section. The bridge mode container networks that are used in this section will be similar to that.

It's also good to note that the veth pairs that were created in the bridge mode network were created with the MTU value (Maximum Transmission Unit) of 1450 bytes because of the VXLAN headers. VXLAN headers add a 50-byte-overhead to every packet delivered through a VTEP [41]. If the veth pairs were not created with the MTU value of 1450 bytes, packets could become too large. This could cause IP fragmentation in the networks which often results in performance penalties and transmission issues. Regular Ethernet or WiFi connections expect the MTU value of 1500 bytes [61].

3.2.1 Ethernet connected hosts

Before implementing an overlay mode container network, the bridge mode container networks need to be set up. This can be done as was described in the sections before. From here on, there will be assumed two hosts referred to as *Host1* and *Host2* that both have two containers in a bridge mode container network (Figure 10).
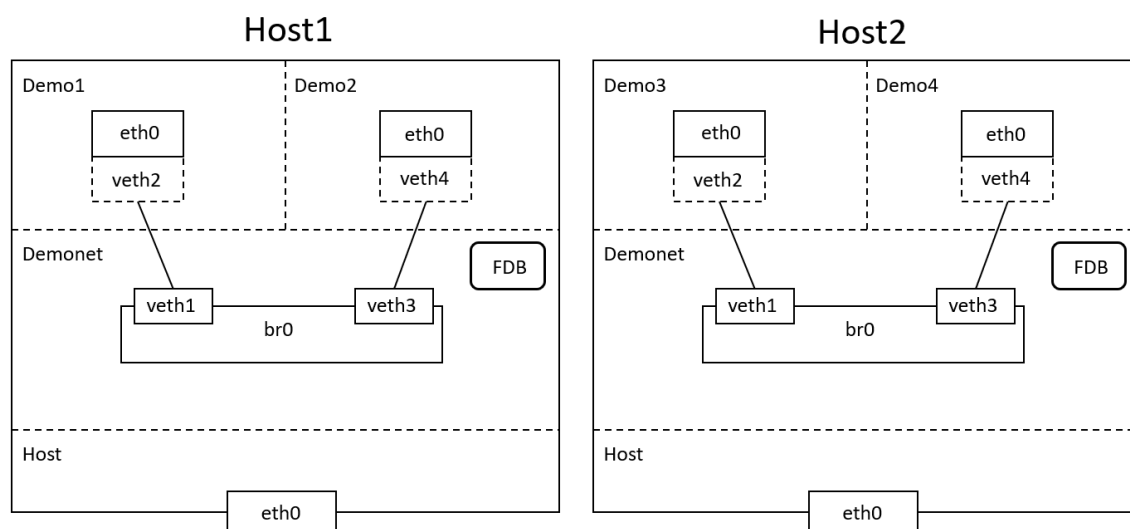

Figure 10: Two hosts with bridge mode container networks. The attached containers are demo1, demo2, demo3, and demo4.

Host1 has been set up exactly as in the sections before. Host2 is set up similarly but the first container is called *demo3* and it has MAC address 02:42:c0:a8:00:04 and IP

address 192.168.0.4. The second container is called *demo4* and it has MAC address 02:42:c0:a8:00:05 and IP address 192.168.0.5. To avoid any potential networking conflicts, it's important to ensure that none of the containers share the same MAC or IP address.

To enable VXLAN tunneling from a bridge mode network, a VXLAN interface is required [62]. This interface must be created in the host's network namespace so that it can create and maintain a link to the host's network interface and send traffic over the network. After the VXLAN interface is created, it can be moved to the bridge mode network and attached to the bridge. This shall be done on both hosts Host1 and Host2 (Figure 11).
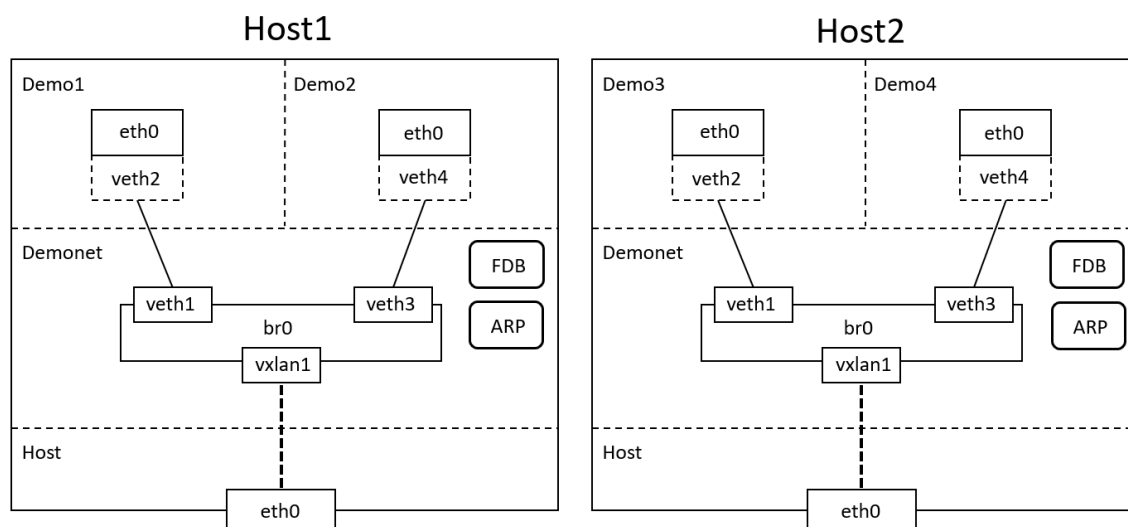

Figure 11: VXLAN interfaces set up on both hosts. VXLAN interfaces maintain an ARP table with MAC and IP address mappings of containers on the other host.

To set up the VXLAN interfaces, the following commands should be run on both hosts:

```
# Create a VXLAN interface with VNI value 42
$ ip link add dev vxlan1 type vxlan id 42 proxy \
$> nolearning dstport 4789

# Move the VXLAN interface to demonet
$ ip link set vxlan1 netns demonet

# Attach the VXLAN interface to the bridge
$ ip netns exec demonet ip link set vxlan1 master br0

# Bring the VXLAN interface up
$ ip netns exec demonet ip link set vxlan1 up
```

When creating a VXLAN interface, it's important to set the correct VXLAN network identifier (VNI). The VNI value is written to every VXLAN header and read by the VTEP. The VNI value tells the VTEP to which network a packet should be forwarded. In this case, packets are forwarded to demonet.

VXLAN interfaces maintain MAC and IP address mappings of containers on remote hosts. If the interface is created with the proxy option, it can work as an ARP proxy and answer ARP queries locally based on these mappings [40]. This allows the local containers to discover containers from remote networks. However, the interfaces don't have any of these mappings by default, but they can be configured manually.

```
# On Host1:
# Create an ARP entry for 192.168.0.4 (demo3 on Host2)
$ ip netns exec demonet ip neighbor add 192.168.0.4 \
$> lladdr 02:42:c0:a8:00:04 dev vxlan1
```

The containers demo1 and demo2 on Host1 can now discover the MAC address of the container demo3 with an ARP request. This ARP request is answered locally by the VXLAN interface. The containers demo1 and demo2 can also create and send network packets to the VXLAN interface. However, the VTEP isn't yet configured to forward these packets to the destination host.

VTEP reads the forwarding table maintained by the network bridge to find out how a packet should be forwarded out in the network [41]. As network bridges work at the data link layer in the OSI model, the tables are indexed by destination MAC addresses [63]. To tell the VTEP how a packet should be forwarded, an entry needs to be added to the forwarding table.

The following command adds the required entry to the forwarding table:

```
# On Host1:
# Forward packets destined to demo3 to demonet on Host2
# through the VXLAN interface
$ ip netns exec demonet \
$> bridge fdb add 02:42:c0:a8:00:04 dev vxlan1 \
$> self dst 10.0.0.3 vni 42 port 4789
```

The added entry defines the IP address and port number to use when connecting to the destination host. On the destination host, the VTEP listens to the port 4789 that is the IANA assigned VXLAN port in RFC 7348 [38]. The VNI value tells the receiving VTEP to which network the packet should be forwarded to.

Host1 can now forward packets from its bridge mode network through a VXLAN tunnel to the VTEP on Host2. The VTEP on Host2 receives these packets and forwards them to its bridge mode network where the packets are eventually received by the container demo3. However, as the MAC addresses of the containers on Host1 are not yet known to the VXLAN interface on Host2, the container demo3 can't respond.
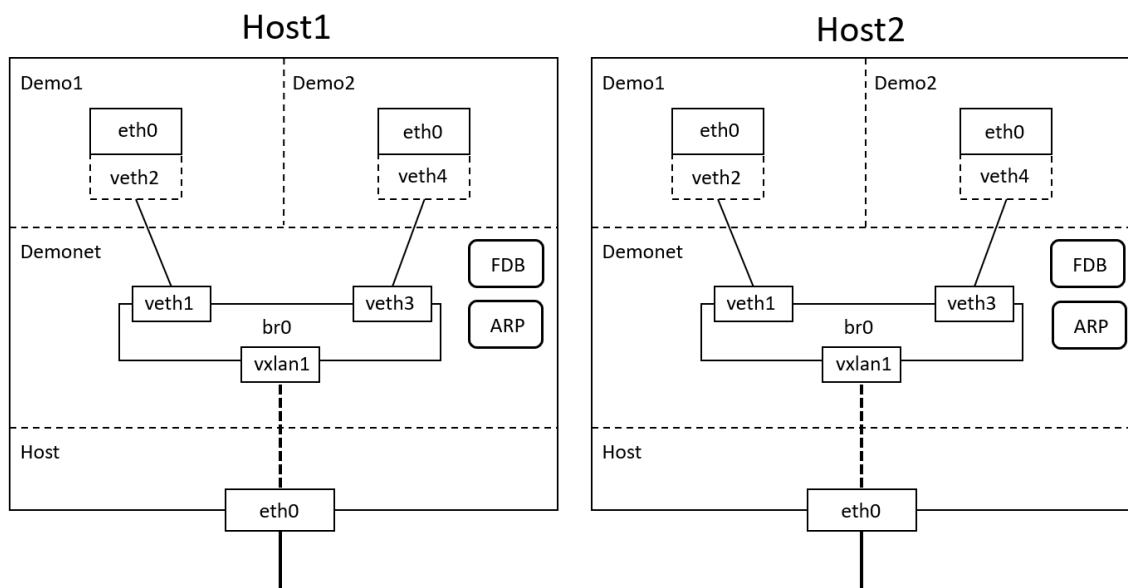
Figure 12: VXLAN overlay network set up. Containers can now communicate across two different hosts.

The VXLAN ARP proxy and forwarding table entry can be set up on Host2 similarly as on Host1 with the following commands:

```
# On Host2:
$ Create an ARP entry for 192.168.0.2 (demo1 on Host1)
$ ip netns exec demonet ip neighbor add 192.168.0.2 \
$> lladdr 02:42:c0:a8:00:02 dev vxlan1

# Make the MAC address of demo1 accessible
# through the VXLAN interface
$ ip netns exec demonet \
$> bridge fdb add 02:42:c0:a8:00:02 dev vxlan1 \
$> self dst 10.0.0.2 vni 42 port 4789
```

The overlay mode network has now successfully been set up in a way that allows communications between containers demo1 and demo3 that are in different bridge mode networks and on different hosts (Figure 12). This can be verified with ping requests. To enable communications to containers demo2 and demo4 in the overlay mode network, the same way of setting up configurations should be followed.

Verify network connectivity between containers demo1 and demo3 with ping requests:

```
# On Host1, ping from demo1 to demo3
$ docker exec demo1 ping 192.168.0.4
> 64 bytes from 192.168.0.4: time=1.08ms
```

```
# On Host2, ping from demo3 to demo1
$ docker exec demo1 ping 192.168.0.2
> 64 bytes from 192.168.0.2: time=1.07ms
```

3.2.2 WiFi connected hosts

In this section, it's shortly demonstrated that VXLAN is agnostic to the underlying physical network. Two hosts can be set up exactly as in the previous section with Ethernet connected hosts. The only difference this time is that the Ethernet interfaces will be brought down so that only WiFi networking is enabled.

To bring down the Ethernet interfaces, the following command should be run on both hosts:

```
# Disable Ethernet interface on a host
$ ip link set dev eth0 down
```

Now, once the overlay network has been implemented with VXLAN, the connectivity can be verified with ping requests:

```
# On Host1, ping from demo1 to demo3
$ docker exec demo1 ping 192.168.0.4
> 64 bytes from 192.168.0.4: time=9.82ms

# On Host2, ping from demo3 to demo1
$ docker exec demo2 ping 192.168.0.2
> 64 bytes from 192.168.0.2: time=9.87ms
```

So VXLAN works over WiFi but the connections can be significantly slower compared to Ethernet. Here the response times are almost 10ms while on Ethernet connected hosts they were close to 1ms. Therefore, in setups where speed is essential, Ethernet should be preferred over WiFi. However, WiFi has its own benefits in being wireless and reducing the cabling complexity.

3.3 VXLAN control plane

In this section, packet delivery with VXLAN is analyzed in more depth. The analysis is made by looking at the overlay mode container networking setup used in the previous section. The VXLAN functionality has been implemented with the Linux kernel tunnel device since Linux 3.12 [37].

To send a packet to another container in the overlay mode network, a container needs the MAC address of the destination container. MAC addresses can be discovered

with ARP requests to the network [64]. However, if the destination container is not attached to the same network, it can't respond to these ARP requests.

The VXLAN interfaces created in the previous section were set up to work as ARP proxies. Using the MAC and IP address mappings, the VXLAN interfaces could respond to ARP requests on behalf of the remote containers that were in other networks (Figure 13). This allows the MAC address of any container to be discoverable across the entire overlay mode network.
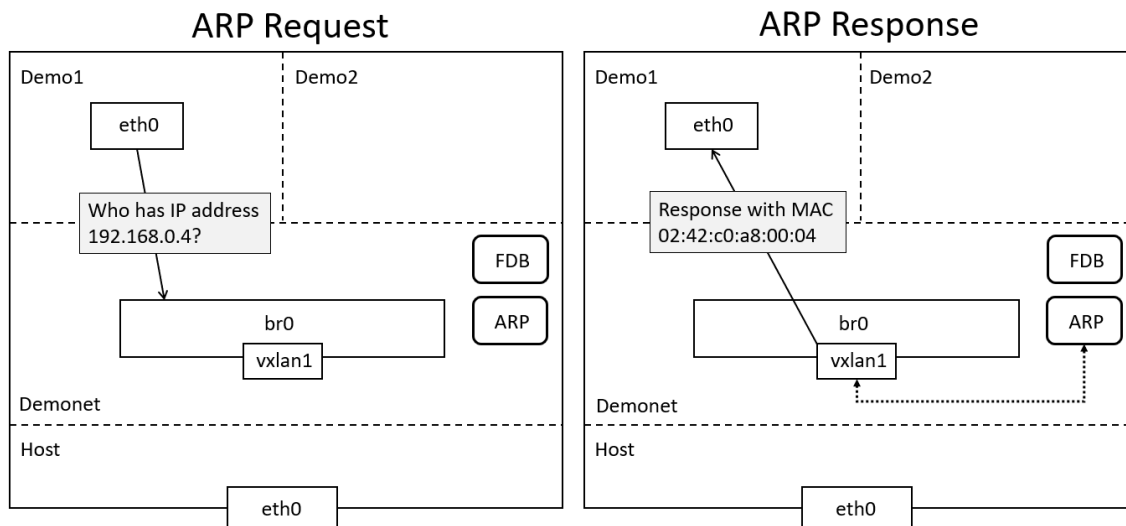

Figure 13: Container demo1 on Host1 sends an ARP request to get the MAC address of demo3 on Host2. The VXLAN interface is working as an ARP proxy and provides the MAC address locally.

Once a container knows the MAC and IP addresses of the destination container, it can start sending packets. If the destination container is in another network, the packets are forwarded in the local network to the VXLAN interface. The packets are then passed to the VTEP that operates in kernel space.

The packets that the VTEP receives are Ethernet frames. To create the VXLAN tunnel, the VTEP encapsulates the frames in IP packets. The IP address of the destination host is searched from the forwarding table and the packets are forwarded to the network through the host's network interface (Figure 14).

Entries in the forwarding table can be viewed with the following command:

```
# Show entries in the forwarding table
$ ip netns exec demonet bridge fdb show
> 02:42:c0:a8:00:04 dev vxlan1 dst 10.0.0.3 link-netnsid 0 self
> permanent
```

To be more specific, the Ethernet frames received by the VTEP are first added a VXLAN header. The VNI value that tells the receiving VTEP to which network
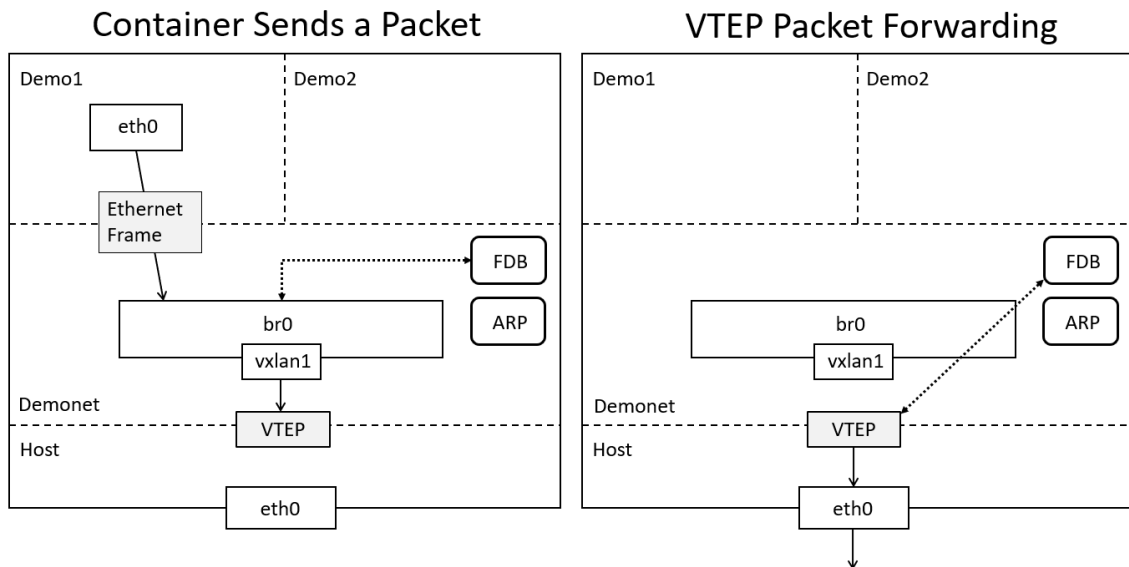
Figure 14: The bridge looks up the forwarding table and forwards the packet to the VTEP through the VXLAN interface. The VTEP looks up the forwarding table, encapsulates the packet, and forwards it to the network through the host's network interface.

the frame should be forwarded to is stored in this header. Then, both the Ethernet frame and the VXLAN header are encapsulated in a UDP packet that then again is encapsulated in an IP packet [65] (Figure 15).

Figure 15: VXLAN packet encapsulation. The UDP source port is dynamic which allows load balancing.

Once the encapsulated packets are received by the destination host, it forwards them to the VTEP. The VTEP decapsulates the packets and reads the VNI values from the VXLAN headers. Then, based on the VNI value the packets are forwarded to the destination network and, finally, to the container.

3.4 Traffic control

Containers that run on the same host also share the resources of this host. With networking, this brings a risk that a single container can consume too many resources and impair networking for other containers. The other containers could experience bad throughput or even lost connections. Therefore, it's important to set restrictions on container network usage.

The Linux traffic control subsystem [66] is a user-space utility that allows configuring the Linux kernel packet scheduler. It can be used to control and shape network traffic on a network interface. For example, using the container networking setup configured earlier, all the packets sent from container demo1 can be added a constant delay of 200ms.

```
# Before adding the delay of 200ms
$ docker exec demo1 ping 192.168.0.3
> 64 bytes from 192.168.0.3: time=0.138ms

# Set 200ms delay on all packets sent from demo1
$ ip netns exec $ctn_ns1 tc qdisc add dev eth0 \
$> root netem delay 200ms

# After adding the 200ms delay
$ docker exec demo1 ping 192.168.0.3
> 64 bytes from 192.168.0.3: time=201ms

# Remove the rule for constant delay
$ ip netns exec $ctn_ns1 tc qdisc del dev eth0 \
$> root netem delay 200ms
```

The traffic control subsystem uses queuing disciplines [67] to implement Linux kernel traffic management capabilities including classification, prioritization, and rate shaping. These effectively allow controlling the way that data is sent. The queuing discipline layer sits between the IP stack and the driver queue of the network interface controller (NIC).

A driver queue contains descriptors which point to socket buffers (sk_buff). Socket buffers are data structures that hold the packet data and are used throughout the kernel. The driver queue exists to ensure that whenever the system has data to transmit, it's available to the NIC for immediate transmission. The IP stack queues complete IP packets to the driver queue from where they are then dequeued by the hardware driver and sent to the NIC hardware for transmission.

To keep the NIC driver software simple and fast, the driver queues are typically implemented as first-in, first-out (FIFO) buffer rings [68]. More complex queue management behaviors can then be carried out on a higher layer that is the Linux abstraction for queuing disciplines. As this layer sits between the IP stack and the driver queue, changes to the queue management don't require changing the IP stack or the NIC driver.

In the context of container networking, the primary objective is to prevent a single container from bloating the host's driver queue. A bloated driver queue would result, for example, in delayed packet transmission for other containers. To prevent this, the network interfaces of the containers need to be slowed down.

Token bucket filter (TBF) is a simple queuing discipline that allows slowing an interface down. It only passes packets arriving at a rate which is not exceeding some administratively set rate, but with the possibility to allow short bursts that exceed this rate. A network throughput measurements tool called *iperf3* can be used to verify that TBF appropriately slows down a container's interface.

```
# Start iperf3 on host
$ iperf3 -s -f K

# Run bandwidth test from network namespace of demo1
$ ip netns exec $ctn_ns1 iperf3 -c 192.168.0.4 -f K -t 5
> 1. 5330724 KBytes/sec
> 2. 5359365 KBytes/sec
> 3. 5306148 KBytes/sec
> 4. 5300170 KBytes/sec
> 5. 5337376 KBytes/sec
> On average 5326757 KBytes/sec sent

# Set rate limiting on demo1
$ ip netns exec $ctn_ns1 tc qdisc add dev eth0 root tbf \
$> rate 123kbit burst 456kbit latency 250ms

# Run bandwidth test from network namespace of demo1
$ ip netns exec $ctn_ns1 iperf3 -c 192.168.0.4 -f K -t 5
> 1. 154 KBytes/sec
> 2. 2.73 KBytes/sec
> 3. 0.00 KBytes/sec
> 4. 264 KBytes/sec
> 5. 0.00 KBytes/sec
> On average 84.1 KBytes/sec sent
```

As can be seen in the iperf3 results, TBF effectively slows down the networking interface of container demo1. It allows temporary bursts but then quickly throttles the traffic to ensure that the exceeded state won't last. Overall, the average throughput remains under the allowed rate.

A similar approach to control traffic with TBF is adopted also for example by Kubernetes. It defines a standardized container networking interface (CNI) that allows configuring networking for containers. The actual implementation is shipped in the CNI bandwidth plugin that, in addition, allows shaping the inbound traffic.

4 Computer clusters and container orchestration

Container orchestration is the process of organizing and running containerized workloads with the appropriate configurations so that applications function as designed.

With hundreds of containers, this process can be fairly inadequate to manage manually. Tools such as Kubernetes and Docker Swarm allow automating this process, and can be used to define container deployments that are then run with automated updates, health monitoring, and failover procedures.

A computer cluster is a set of interconnected computers that work together in a way that appears as a single computer to the end user. One of the main benefits of computer clusters over single computers is their approach to high availability. They attempt to eliminate single points of failure by having redundant computers. If any of the computers would fail, its workloads can be rescheduled on other computers in the cluster to ensure the provision of services without interruption.

In terms of scalability, the performance, redundancy, and fault tolerance of a cluster improves horizontally with the addition of new computers. This allows for cost effective improvement compared to scaling up a single computer. With load balancing, in which smaller computational workloads are distributed across multiple computers, a large number of lower performing computers can become a "supercomputer" that executes large computational workloads.

4.1   Kubernetes

Originally developed by Google, and later donated to the Cloud Native Computing Foundation (CNCF), Kubernetes has become today the world's most popular container orchestration tool [69]. It allows interconnecting individual computers into a cluster that can be used as a platform where all its components, capabilities, and workloads are configured. It's designed to manage the entire life cycle of containerized applications with automated operations such as scaling, distributing, and fault handling.

The core, or the brain, of a Kubernetes cluster is a data store called the cluster state. It describes both the current and desired state of a cluster. When these two diverge, the Kubernetes components automatically detect this divergence and make the required changes to move the current state towards the desired state. This is how a Kubernetes cluster, for example, recovers from failures. A user can also modify the desired state to deploy new applications or to change the current behavior of the cluster.

Computers constituting a Kubernetes cluster, are each assigned to a role. This role is either a master node or a worker node. Worker nodes are sometimes also referred to as just nodes. The role assigned to a computer is static, meaning that a master node can't become a worker node and vice versa. The duties of master and worker nodes are also clearly separated with both of them being responsible for a very different set of tasks.

Master nodes, in general, are responsible for maintaining the desired state of a cluster [70]. They expose an API for users and clients, health check other nodes, decide how to split up and schedule work, and orchestrate communications between

the different components. A cluster can have one master node, or more for better availability and redundancy.

In clusters with more than one master node, one of them is elected as a so called leader master. Leader master is the centralized authority in a cluster and the primary responsible for managing the cluster's state. It's followed by the other master nodes, and if it dies, one of its followers is elected as the new leader master.

A Kubernetes master node is a collection of three processes typically run on a single node. These three processes are *kube-apiserver*, *kube-controller-manager*, and *kube-scheduler*. In addition to these processes, the data store that contains the cluster's state is also a component of the master node.

Worker nodes are responsible for performing the actual work in a cluster. They request work instructions from master nodes and then perform the work accordingly. As worker nodes are primarily just computational resources controlled by master nodes, users very rarely need to interact with them directly [70] (Figure 16).
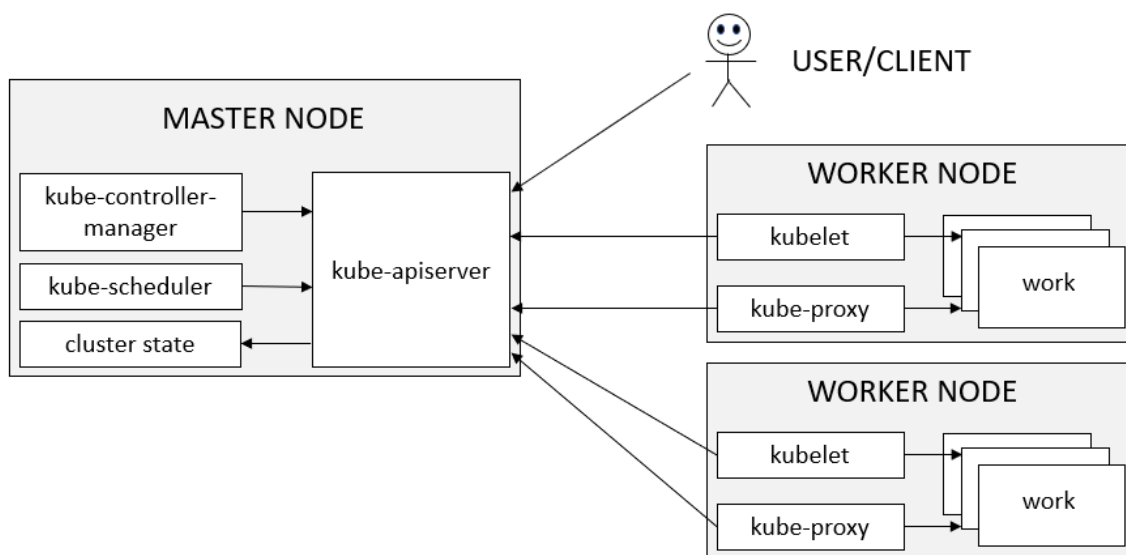

Figure 16: Kubernetes cluster.

The two primary processes on a worker node are *kubelet* and *kube-proxy*. Kubernetes also implements a Container Runtime Interface (CRI) [71] that allows easy adoption of different container runtimes. The most popular of these is Docker.

While Kubernetes comes with a rich set of different features, its capabilities are often extended with various plugins. Kubernetes has multiple layers of abstraction that provide these plugins with interfaces they can use to interact with the state of the cluster. One such interface is for example the Container Network Interface (CNI) [72] that is a CNCF specification for how plugins can configure network connectivity for containers and containerized applications [73].

4.1.1   API server

The Kubernetes API server, kube-apiserver, is the primary management point of the entire cluster that users, processes, and components communicate with [74]. It's also the only Kubernetes component that directly interacts with the data store that contains the cluster state. Therefore, any operation that interacts with or configures the Kubernetes workloads or objects must pass through the API server.

The API server provides a REST API that supports the common CRUD (create, retrieve, update, and delete) operations via the standard HTTP verbs GET, POST, PUT, and DELETE with JSON as the default payload format. Users usually utilize these operations through a client program called *kubectl*. Kubectl is a command-line tool that comes built-in with Kubernetes.

4.1.2   Cluster state

The state of a Kubernetes cluster is described through abstractions called Kubernetes objects. These objects are persistent entities that maintain the cluster configurations and state information. The state information can be split to current state and the desired state information.

The current state describes such as which applications are running on which nodes, the resources available to them, and how they can be communicated with. The desired state describes then such as how many replicas of a given application should be running and how much resources should be available to each replica. The objects can also be configured different policies for such as restarts, upgrades, and fault tolerance.

The Kubernetes objects are stored to a data store that by default is etcd. Etcd [75] is a distributed key-value store. In clusters with multiple master nodes, it's responsible for ensuring that the cluster state stored on every master node is consistent and up to date.

The other Kubernetes components access the cluster state through the API server. In fact, most of the components communicate with one another only through the cluster state. They regularly check for a particular object to see if its current state matches its desired state. If these two states diverge, they make modifications that are then acknowledged by other components, and so on, until the current states of all the objects match their desired states again. Because the cluster state plays such a central role in a cluster, it's often referred to as the cluster's shared state.

The basic Kubernetes objects include:

Pod - Pod [76] is the basic execution unit of a Kubernetes application and represents a process running in the cluster. It encapsulates one or more application containers, their storage resources, and has a unique network IP. It's defined with options that govern how the container(s) should run. Kubernetes scales applications up horizontally by creating new instances of the application pod.

Service - Service [77] is an abstraction which defines how a set of pods can be accessed. As pods are mortal and a set of pods running an application in one moment can be different from the set of pods running the application a moment later, a service is necessary for keeping the pods accessible to clients.

Volume - Volume [78] is an abstraction of a directory accessible to containers in a pod. When a container dies, all its on-disk files are lost upon restart. However, with volumes files can be persisted over container restarts.

Namespace - In Kubernetes terms, namespaces [79] do not refer to Linux namespaces, but are a way to divide resources between multiple users, projects, or teams. For example, pods running in one namespace can be allocated a different amount of resources than to pods running in another namespace.

In addition to the basic Kubernetes objects, there are also a number of higher-level abstractions called controllers. Controllers build upon the basic objects, and provide additional functionalities. They include:

ReplicaSet - ReplicaSet [80] ensures that a specific number of pod replicas are running at any given time in a cluster. If a pod crashes, ReplicaSet ensures that it will be recreated to get back to the desired state.

Deployment - Deployment [81] provides declarative updates for pods and ReplicaSets by changing the desired state. For example, a deployment can be used to create a new ReplicaSet, remove the existing deployment, and adopt the resources with the new deployment.

DaemonSet - DaemonSet [82] is used to define which nodes should run a copy of a pod. For example, it can be used to define that all nodes should run a copy of a monitoring or log collection application pod.

4.1.3 Controllers

The Kubernetes controller manager, kube-controller manager, is a process that watches the cluster state through the API server [83]. It manages workload life cycles, performs routine tasks, and in general, regulates the state of a cluster. To do this work, it runs a so called control loop.

The control loop consists of a group of controllers where each controller is responsible for watching after a particular Kubernetes object. The controllers watch for such as creation, modification, and deletion events. Whenever a controller notices that the current state of an object diverges from its desired state, it makes changes attempting to move the current state towards the desired state.

A controller could be imagined as the following loop:

```
while True:
    desiredState = object.getDesiredState()
    currentState = object.getCurrentState()
    if (desiredState != currentState):
```

```
makeChanges(object)
```

For example, the node controller is responsible for all the nodes in a cluster. If a node goes down, it notices the changed state and schedules the pods on that node for deletion. When the pods are deleted from a node, this event is caught by the replication controller that is responsible for maintaining the correct number of pods running in a cluster. The replication controller then reschedules the deleted pods on other nodes to ensure that the desired number of pods are running in the cluster.

Most of the cluster-level functions in a cluster are performed by the controllers. These functions include such as garbage collection, API business logic functions, and life cycle management. The controllers also manage the application management and composition layer providing such as self-healing, scaling, service discovery, routing, and service binding and provisioning.

To minimize the performance overhead, the controllers are level driven. This means that instead of constantly requesting an object's information from the API server, they use event hooks to receive notifications of adds, updates, and deletes for a particular object. This allows any controller to be off for an indeterminate amount of time reacting only for changes to relevant objects.

Nearly all of the controllers can be collapsed into a queue of "check this X" based on relationships. This has allowed splitting each controller into a separate component that is easy to extend or replace. Together, these components then work in a decentralized and decoupled choreography-like coordination without a message bus.

4.1.4   Scheduler

The Kubernetes scheduler, kube-scheduler, is an independent process that's job is to ensure that all pods are assigned to worker nodes [84]. It constantly watches for pods through the Kubernetes API and is actually kind of a controller even that it's not managed by the Kubernetes controller manager. Whenever a pod that is not assigned to any worker node is found, the scheduler enters a node selection process to assign the pod to a worker node. Without the kube-scheduler, pods wouldn't be assigned to any worker nodes in the cluster and would remain in a pending state indefinitely.

The scheduler could be imagined as the following loop:

```
while True:
    pods = getAllPods()
    for pod in pods:
        if pod.node == nil:
            assignNode(pod)
```

In the node selection process, the scheduler tries to find the most suitable node for the pod. If there are no suitable nodes, the pod remains unscheduled until a node

becomes suitable. A suitable node is such that the requested resources of the pods already running on the node plus the requested resources of the new pod are not greater than the capacity of the node. Pod specifications may also set additional requirements and conditions for suitable nodes that the scheduler checks during the node selection process.

If multiple nodes are suitable for the pod, the scheduler will rank the nodes to find the most suitable. Typically the most suitable node is the one whose already running pods consume the least resources. This ensures that pods are spread out evenly without packing them onto some nodes while leaving others empty. After the scheduler has decided the most suitable node for the pod, the pod is assigned to this node.

However, the pod is not run yet at this point. The kube-scheduler is responsible only for making sure that every pod is assigned to a worker node. It's the kubelet's responsibility to then read this assignment and actually run the pod on a worker node.

When the scheduler watches for pods in a cluster, it uses event hooks just as the other Kubernetes controllers. This allows avoiding the overhead that would be caused if the scheduler queried all the pods in the cluster every time it needed to schedule a new pod. In Kubernetes, the event hooks are actually called informers and they provide hooks to receive notifications of adds, updates, and deletes for a particular object.

4.1.5   Kubelet

Kubelet is the primary process running on each worker node and responsible for performing the actual work in the cluster. It polls the API server regularly to watch for changes in work instructions and adjusts the work based on what it finds. Mostly this means launching new pods assigned to it, or killing those that have been removed from it.

Once a pod has been assigned to a worker node, the kubelet running on that node becomes responsible for ensuring that the pod is running and healthy. Kubelet runs its own control loop to ensure that the current state of a pod matches the desired state. The control loop includes such as life cycle hooks and health and readiness checks on containers running in pods, while adhering to the restart policies of these pods. Kubelet manages the pods through the Container Runtime Interface (CRI) [85].

```
// Container Runtime Interface (CRI)
service RuntimeService {

    // Sandbox operations.
    rpc RunPodSandbox(RunPodSandboxRequest)
    rpc StopPodSandbox(StopPodSandboxRequest)
```

```
        rpc RemovePodSandbox(RemovePodSandboxRequest)
        rpc PodSandboxStatus(PodSandboxStatusRequest)
        rpc ListPodSandbox(ListPodSandboxRequest)

        // Container operations.
        rpc CreateContainer(CreateContainerRequest)
        rpc StartContainer(StartContainerRequest)
        rpc StopContainer(StopContainerRequest)
        rpc RemoveContainer(RemoveContainerRequest)
        rpc ListContainers(ListContainersRequest)
        rpc ContainerStatus(ContainerStatusRequest)

        ...
    }
```

When kubelet finds that it has been assigned with new work, that is to run a pod, it requests the pod specifications from the API server. Before starting the pod, kubelet calls the RunPodSandbox function in the CRI to create an isolated environment with resource constraints for containers. This environment is set up with networking including such as an allocated IP address.

Once active, the environment will work as the pod. Individual containers can be created, started, stopped, and removed within it by calling the CRI. The pod is then started by running within it the containers that are defined in the pod specification. After this, Kubelet will regularly update the status of the pod to the cluster state by calling the API server.

Kubelet can also reject pods that have been assigned to it. For example, if the worker node is out of memory, kubelet can set the state of a pod to failed through the API server. The failed pod is caught by the control loop run on the master and eventually the pod is returned back to the scheduler. This makes kubelet the final arbiter of what pods can run on a given node.

4.1.6 Container Runtime Interface

Originally Kubernetes supported only Docker and rkt as its container runtimes and they were integrated deeply into the kubelet source code. This made integration of other container runtimes difficult as the process required a deep understanding of the kubelet internals. To allow a larger variety of container runtimes, that each have their own strengths, the Container Runtime Interface (CRI) was developed. It provides a clearly defined interface for plugging any container runtime.

The CRI listens for requests from the Kubelet and provides the container runtime operations through three standardized interfaces that are wrapped inside the CRI [86]. These interfaces are OCI Image Format, OCI Runtime, and Container Network Interface (CNI) [87]. Each of these interfaces can be plugged with a different plugin.

The OCI Image Format specification defines how to build, transport, and prepare a container image to run. A container image is like a snapshot of a container containing metadata about its contents and dependencies. Once built, the container image can be unpacked into a runnable filesystem bundle that the runtime can consume.

The OCI Runtime specification defines the configuration, execution environment, and life cycle of a container. At a high level, the runtime takes a container image, reads its configurations, unpacks it into a runnable filesystem bundle, and executes the bundle in a consistent environment, that is the container. The runtime also allows setting different event hooks that are invoked over the timeline of events that happen from when the container is created to when it ceases to exist.

The CNI specification defines how to configure networking for containers, where container is considered synonymous with a Linux network namespace. In Kubernetes, this unit corresponds to a pod. The CNI concerns itself only with network connectivity of containers and removing the allocated resources when a container is deleted.

The CNI interface is different from the other two interfaces in that it allows running multiple CNI plugins for a single container. The plugins are executed sequentially, each plugin being responsible for inserting a network interface into the container's network namespace and making any necessary changes on the host. The CNI also implements a second type of a plugin called the IP Address Management Plugin (IPAM) that the other plugins can invoke to determine the network interface's IP address and subnet, gateway, and routes.

4.1.7   Kubenet

Kubelet has only a single network plugin by default that is called kubenet [88]. Kubenet is registered with the CNI interface when kubelet starts. It's a very simple CNI plugin that doesn't implement advanced networking features such as cross-node networking or pod network policies. It only creates pods an isolated local area network on each host.

Kubenet sets up the local area network on a worker node by first creating a Linux bridge. This is done before any pods are run on the host. Then, when pods are created, they are attached to this bridge through veth pairs. Kubenet operates similar to the Docker default bridge mode networking.

When a new worker node joins the cluster, the controllers run on the master allocate it a range of IP addresses. This range is by default a /24 CIDR block (256 addresses) and corresponds to the maximum number of pods per node. The block is also different for each node to ensure that the pod IP addresses are unique across the cluster.

To create a new pod, kubelet calls the CRI that passes the call eventually through the CNI to kubenet. Kubenet attaches the pod's network namespace to the Linux bridge with a veth pair and assigns the pod end of the pair an IP address from the

IP addresses allocated to the node. This IP persists with the pod until it's deleted.

IP addresses in Kubernetes exist at the pods' scope. Every pod is given its own unique IP address that is the so called "IP-per-Pod" model. Containers within a pod share their network namespace - including the IP address - allowing them to reach each other's ports through localhost. This makes a pod much like a virtual machine or a physical host from the perspectives of port allocation, naming, service discovery, load balancing, application configuration, and migration.

It's also important to note that a pod itself does not run. It's just an environment in which containers are run. Therefore, even if containers are restarted in a pod, the pod itself - including its IP address - doesn't change throughout its life cycle.

4.1.8 Kube proxy

The Kubernetes network proxy, kube-proxy, runs on every worker node [89]. It's responsible for enabling network communications for pods in a Kubernetes cluster. It's a controller, like many other controllers in Kubernetes, and watches the API server for changes and makes updates accordingly.

The name of kube-proxy is misleading in that it's not an actual proxy. It used to be an actual proxy until the Kubernetes version 1.0 but this implementation turned out to be resource intensive and slower due to constant copying between kernel and user space. In the newer versions of Kubernetes, kube-proxy enables networking by using iptables rules.

The functionality of kube-proxy is closely tied with the Kubernetes service objects. When a new service is created, the Kubernetes master assigns it a stable and reliable virtual IP address from the cluster's pool of available IP addresses [90]. The assigned IP address is unique within the cluster and doesn't change throughout the life cycle of the service. The IP address is only released if the service is deleted from the cluster's configuration.

A service object also defines a group of pod IP addresses as endpoints that are accessible through its virtual IP address. This allows decoupling the access logic to pods from their actual IP addresses. The controller for the service continuously scans for pods that match its selector, and if the IP address of a pod has changed, the endpoint is updated accordingly [91].

When kube-proxy notices a change in service objects, it reads the specifications of the new or changed service through the API server and updates the iptables rules accordingly. The rules are set so that when a packet is destined to the virtual IP address of a service, the destination IP is changed to one of the defined pod IP addresses. This way loads are distributed evenly among groups of pods and the virtual IP address of a service can act also as a load balancer.

The iptables rules are set to use DNAT (Destination Network Address Translation) for packets that are destined to a virtual IP address of a service. This means that not only the destination IP address is changed, but this translation is also stored to the

Linux connection tracking table. The stored translation is a 5-tuple that contains the protocol, source IP address, source port, destination IP address, and destination port of a packet. Then, when a reply comes back later, the stored translation is used to change the source IP address of the response packet to the virtual IP address of the service. This makes the packet flow transparent to clients.

For packets that are destined to external services outside the cluster, the iptables rules are set to use SNAT (Source Network Address Translation). This tells the kernel to use the IP address of the interface a packet is going out from (node IP address), instead of the pod's. Without this translation, packets would be sent out with the pod's IP address that is private and not known to routing systems outside the cluster. This would result in that the outside networks wouldn't know how to route a response packet and it would be dropped before reaching the node on which the pod resides.

In Kubernetes, a service can also be made available to external clients by setting it the type NodePort. This assigns the service an actual port in range 30000-33000 that is open to public on every node, even if there's no pod running on a particular node. Inbound traffic to this NodePort would then be sent to one of the pods (that may even be on another node) with, again, iptables.

4.1.9   Journey of a container

An application can constitute of one or more containers. The containers are defined in the OCI image format and stored to an image registry. This registry is utilized by the cluster to pull and unpack these images and run as containers.

To run an application, a user calls the API server to define a new Kubernetes deployment object. The deployment object defines a set of containers that should be run in a pod and the number of replicas of this pod that should be deployed in the cluster. This is also known as the desired state of the deployment and it's stored to the shared state of the cluster.

Deployment controller watches for deployment objects through the API server and is responsible for ensuring that the desired state of a deployment matches its current state in the cluster. When it finds the new deployment, it reads the pod specifications and creates a new pod template. It also reads the defined number of pod replicas and stores the results as a new ReplicaSet object.

ReplicaSet controller watches for ReplicaSet objects through the API server and is responsible for ensuring that there is the correct number of pods running in the cluster. When it finds the new ReplicaSet, it reads the pod template and creates the desired number of pod objects.

Scheduler watches for pod objects through the API server and is responsible for ensuring that they all are assigned to a node. When it finds the new pod objects that haven't yet been assigned to any node, it enters a node selection process. Once suitable nodes have been found, the pods are assigned to nodes by tagging the pod

objects with node labels.

On each node, kubelet watches for work instructions through the API server and is responsible for running the pods assigned to it. When kubelet finds a new pod assignment, it requests the pod specifications from the API server and downloads the container image(s) from the image registry. It then calls the CRI to create the pod and run the one or more containers in it. The created pod is also assigned a pod IP address from the node's pool of available IP addresses. Once the pod has become operational, kubelet updates the state of the pod object, including such as the pod IP address, in the shared state of the cluster by calling the API server.

As pod IP addresses are unreliable for directly being used in communications, the user can call the API server to create a new service object with an access policy to the pods. The new service defines a pod selector and is assigned a virtual IP address from the cluster's pool of available IP addresses. The controller for the service then continuously scans for pods that match its selector, and updates any changed pod IP address.

On each node, kube-proxy watches for service objects through the API server and is responsible for managing network communications for pods. When it finds the new service, it reads its access policies and installs them as a series of iptables rules. These rules then allow a client to use the virtual IP address of a service to connect to the correct pod(s).

4.2 Consensus

Consensus is a fundamental problem of computer clusters. For example, if master nodes in a Kubernetes cluster lose the consensus about the cluster state, the cluster may stop working. The cluster state in Kubernetes is stored to a distributed data store that by default is etcd. Etcd uses an implementation of an algorithm called Raft for consensus resolution.

Raft [92] is a leader based consensus algorithm to manage a replicated log. It supersedes another consensus algorithm called Paxos. Because Paxos suffered from lack of detail, impracticality, and difficulty in its implementation, Raft was designed to be easily understandable and provide a complete and practical foundation for system building.

In Raft, nodes are divided to a single distinguished leader and its followers. Consensus is implemented by first electing the leader and then making it completely responsible for managing the replicated log. The leader and its followers must always form a majority of the cluster nodes to make progress. For example, if two nodes have failed in a cluster of five nodes, a leader can be elected and the cluster can make progress. However, if more than two nodes have failed the cluster stops making progress as a majority is no longer possible.

To maintain its leadership, a node must send so called heartbeats to all its followers in the cluster. If a follower doesn't receive a heartbeat from the leader for a period

of time, it assumes the leader has failed, and becomes a candidate for a new leader. As a candidate, the node starts an election by sending a vote request to every other node in the cluster. If it receives votes from a majority of the cluster nodes, it becomes the new leader. To establish its authority and to prevent new elections, the node then sends a heartbeat to all the other nodes in the cluster.

Because in Raft a majority of nodes is required for the cluster to progress, the nodes should always be deployed in odd numbers. With Kubernetes this corresponds to the number of master nodes [93]. For example, a cluster of six nodes can survive a failure of two nodes by forming the majority with the remaining four nodes. However, also a cluster of five nodes would survive the failure of two nodes as it can form the majority with just three nodes. Therefore, a new node added to an odd number of nodes doesn't make it any easier to find a majority but is only a new possible point of failure.

When the cluster has a leader with its followers that together form a majority in the cluster, the leader can start progressing and replicating the logs. To write a new entry to the logs, the leader first attempts to replicate it to all its followers. The entry cannot be committed before a majority of the cluster nodes have confirmed the replication. This step includes also other conditions that must be fulfilled, but are left out here for simplicity. Once all the conditions are fulfilled, the new log entry is committed to the logs and it becomes durable. If any of the followers fail the replication, the leader retries it indefinitely until it succeeds.

As Raft requires a majority of nodes to progress, Kubernetes clusters can tolerate up to (N-1)/2 permanent failures, where N is the number of masters. When the number of permanent failures goes beyond this limit, a cluster can no longer process requests that change the cluster state. This means that the existing tasks keep running but nodes cannot be added, updated, or removed, and new or existing tasks cannot be started, stopped, moved, or updated. A cluster can recover from any number of temporary failures.

In Kubernetes, the Raft leader is also elected as the leader master. The leader master works as a centralized authority in a cluster, and is the primary responsible for managing the cluster state. If the leader master dies, one of the other master nodes will become the new leader master. Because Raft ensures that the cluster state replicated on all the master nodes is consistent, the new leader master can pick up the tasks and restore the services in the cluster to a stable state.

4.3 Heartbeat

Heartbeat is a signalling protocol used for monitoring the nodes in a Kubernetes cluster. In a given time interval, every node sends a heartbeat to the master to confirm that its still alive. If a node fails to send a heartbeat before the eviction timeout, it will be considered unreachable and the master will reschedule all its tasks on other nodes to ensure that the cluster remains operational.

The kube-controller-manager manages a controller called the node controller that is responsible for maintaining an internal list of available nodes in the cluster. This list is stored in the cluster state and it represents each node as a node object that contains information such as IP address, healthiness, capacity, and the time of the last received heartbeat [94]. Every heartbeat contains information about a node's state, and the received heartbeats are used to update the node objects in the cluster state.

The default eviction timeout in Kubernetes for a heartbeat is 40 seconds after which the node is considered unreachable and its status is updated to condition unknown. The default value before a master starts evicting the pods assigned to a node with the status condition unknown is 5 minutes. This decision is later communicated back to the node that was unreachable if the communications are re-established. The node controller checks the state of each node every 5 seconds by default.

Nodes might also fail to send the heartbeat signals because the master itself has networking problems. This has been taken into account in the node controller's pod eviction logic. The node controller limits the eviction rate by default to 1 pod per 10 seconds. If the number of unhealthy nodes reaches a configured threshold then the eviction rate is reduced. In small clusters when all nodes have become unhealthy, the evictions can even be completely stopped until some connectivity is restored.

Choosing an optimal eviction timeout for the heartbeat signals is in general difficult. If the timeout is too small, it creates too much overhead. If the timeout is too large, performance degrades as everything waits for the next heartbeat.

Starting from Kubernetes 1.13, a new feature called node lease has been introduced to improve the heartbeat scalability and performance [95]. When enabled, each node is associated with a lease object in the cluster state that is renewed by the node periodically. Node lease is treated as a heartbeat, but is lightweight and significantly cheaper than the actual heartbeat that contains the node state information. Nodes send the node lease frequently while the actual heartbeat is only sent when there is some change or enough time has passed. By default, node lease is sent every 10 seconds and heartbeat every 1 minute.

5   Discussion on challenges

Containers are a lightweight alternative to virtual machines. Applications can be packaged with their execution environment, including runtime dependencies, into container images. As container images have a standardized format, they are portable and fast to deploy on any Linux system with a container runtime.

The downside of being lightweight is that containers provide less isolation than virtual machines that ship with their own virtual hardware. Containers share the kernel and hardware of the underlying Linux system and are isolated only with the Linux built-in isolation tools such as namespaces and cgroups. This means that there's a higher risk for a container interfering with other containers or the

underlying host. With Docker, for example, the default is to run containers in privileged mode with unlimited resources. So if a process escaped from a container it would also be privileged on the host system, or a noisy process could eat up all the resources of the host system denying service from other containers.

The built-in Linux kernel tools allow creating a number of different networking environments for containers. The main difference between these environments is the level of isolation they can provide. The options vary from host mode networking to overlay mode networking. In host mode, there's no isolation and the container shares the network stack of the host. In overlay mode, the containers are connected to an isolated bridge that can then be interconnected with other isolated bridges on other hosts with VXLAN.

One interesting aspect of container networking is that most, if not all, networking can take place within a single host. Instead of packets being transferred through a physical Ethernet cable or broadcast over WiFi, they can be transferred within a Linux system over various virtual network devices. This has a significant impact on how the communications can be restricted or monitored.

In classic hardware-based environments networking is typically host-to-host and packets traverse through a firewall that either allows or blocks traffic based on IP addresses and ports. However, with containers host-to-host traffic usually serves only as a tunnel for container-to-container communications. Especially if the host-to-host tunnel is encrypted, any intermediate firewall won't be able to recognize the communicating containers, and thus, decide whether to allow or block a packet.

Because container-to-container traffic might not even leave a host, or would anyway be hidden inside a host-to-host tunnel, container networking cannot be controlled or monitored from outside the hosts. Therefore, container networks need to be both controlled and monitored from within the hosts. This is not a simple task and requires both understanding and effort.

Container orchestrators such as Kubernetes can be used to run containers in a cluster of interconnected computers. They automate many of the management tasks and continuously monitor for the cluster to ensure that its current state matches the desired state. If the current state diverges from the desired state, they make the necessary changes so that the states match again.

Taking on Kubernetes has a steep learning curve. It's not only a complex system, but also has many moving parts. This is not because of it being poor design, but because it's built to solve even more complex problems in distributed computing.

The popularity of Kubernetes is rapidly growing as more and more organizations are adopting it to their use. However, this technology is still fairly new and subject to continuous change. Blog posts, online guides, and various documentation quickly become outdated making troubleshooting, learning, and even keeping up with the technology more difficult.

According to a StackRox study in 2019 [96], nearly all (94%) of the respondent companies admitted to experiencing a security incident with Kubernetes in the last

12 months. In 69% of these cases the incident had occurred due to a user-driven misconfiguration. The companies cited internal skills shortage and steep learning curve as the two most significant Kubernetes challenges impacting them.

Conceptually, Kubernetes could be thought of as a set of standardized interfaces that allow extending its capabilities with various plugins. This extensibility is one of its strengths but also adds to its complexity. The default plugins that Kubernetes ships with often need to be replaced in real world deployments - and each plugin comes with its own configuration and management overhead.

For example, the default network plugin, kubenet, doesn't allow creating network segments or policies. It places all pods in a single network and allows all connections between them. The pods can even communicate with such as the API server, kubelet, or host nodes. Therefore, it's often replaced with another plugin such as Calico, Weave, or Flannel to enable more advanced networking features.

The default plugins shipped with Kubernetes are not actually even designed to solve all the problems in modern real world deployments. The primary aim of Kubernetes is, instead, to provide such set of standardized interfaces that allows other providers to develop plugins that address these problems. For example, the Container Network Interface (CNI) specification describes how to write plugins to configure network interfaces for Linux containers and defines a common interface between the network plugins and containers.

Networking in general differentiates in Kubernetes from traditional networking. Many of the things that have usually been static, are now dynamic. Such as IP addresses and ports can't be relied on. A pod running in one moment of time might run on another node with a new IP address a moment later.

Because of the dynamic nature of Kubernetes clusters, managing networking configurations on a system level is not feasible. Any static configuration can expire any moment, and the configurations would also need to be applied on every node in a cluster. Therefore, all configurations should be managed through the shared state of the cluster that is stored on the master node. These configurations must also be read, understood, and applied by appropriate plugins to take effect. For developers, this new paradigm might require a change of mindset and adoption of new principles and practices.

Overall, Kubernetes can be a great solution for managing computer clusters with containerized workloads. It's flexible and can be extended with plugins to meet most requirements. However, it's also a complex system that comes with a steep learning curve.

6   Conclusions

Container networks can be built in a number of different ways. These networks are fairly different from traditional physical networks. Managing these networks,

especially in a secure way, can quickly become an issue if the Linux kernel networking subsystem is not understood. Setting up communications restrictions, policies, and network monitoring is not straightforward in a virtualized environment.

Using an automation tool, such as Kubernetes, can be helpful for managing clusters with a large amount of containers. However, Kubernetes comes with a steep learning curve and stacks a number of different layer on top of one another, bringing the end user even further away from the actual networking implementation. Adding the numerous plugins that can be adopted in a Kubernetes cluster, it's easy to lose awareness of what's really going on under the hood.

References

1 "What is a virtual machine?." `https://azure.microsoft.com/en-us/overview/what-is-a-virtual-machine/`.

2 "Resource utilization scheme of idle virtual machines for multiple large-scale jobs based on openstack."

3 A. Grattafiori, *Understanding and Hardening Linux Containers.* NCC Group Whitepaper, version 1.1 ed., June 2016.

4 "Cgroups, linux programmer's manual." `http://man7.org/linux/man-pages/man7/cgroups.7.html`, March 2019.

5 "Namespaces, linux programmer's manual." `http://man7.org/linux/man-pages/man7/namespaces.7.html`, March 2019.

6 "Cve-2019-5736." `https://www.cvedetails.com/cve/CVE-2019-5736/`.

7 "Kernel space definition." `http://www.linfo.org/kernel_space.html`.

8 "Glibc and the kernel user-space api." `https://lwn.net/Articles/534682/`.

9 "Capabilities (7) linux programmer's manual." `http://man7.org/linux/man-pages/man7/capabilities.7.html`.

10 S. Hykes, "Docker 0.9: introducing execution drivers and libcontainer." `https://blog.docker.com/2014/03/docker-0-9-introducing-execution-drivers-and-libcontainer/`, March 2014.

11 "package libcontainer." `https://godoc.org/github.com/docker/libcontainer`.

12 "Libcontainer overview." `http://jancorg.github.io/blog/2015/01/03/libcontainer-overview/`.

13 "Docker leads oci release of v1.0 runtime and image format specifications." `https://blog.docker.com/2017/07/oci-release-of-v1-0-runtime-and-image-format-specifications/`.

14 "Docker overview." `https://docs.docker.com/engine/docker-overview/`.

15 "namespaces - overview of linux namespaces." `http://manpages.ubuntu.com/manpages/bionic/man7/namespaces.7.html`.

16 "pid_namespaces - overview of linux pid namespaces." `http://manpages.ubuntu.com/manpages/bionic/man7/pid_namespaces.7.html`.

17 "Pid_namespaces (7) linux programmer's manual." `http://man7.org/linux/man-pages/man7/pid_namespaces.7.html`.

18 "mount_namespaces - overview of linux mount namespaces." `http://manpages.ubuntu.com/manpages/bionic/man7/mount_namespaces.7.html`.

19 E. W. Biederman, "Ip-netns, linux programmer's manual." `http://man7.org/linux/man-pages/man8/ip-netns.8.html`, January 2013.

20 "user_namespaces - overview of linux user namespaces." `http://manpages.ubuntu.com/manpages/bionic/man7/user_namespaces.7.html`.

21 "The osi model: An overview." `https://www.sans.org/reading-room/whitepapers/standards/paper/543`.

22 "The osi model: Understanding the seven layers of computer networks." `http://ru6.cti.gr/bouras-old/WP_Simoneau_OSIModel.pdf`.

23 "Tcp/ip tutorial and technical overview." `https://www.redbooks.ibm.com/redbooks/pdfs/gg243376.pdf`.

24 "Tcp/ip tutorial and technical overview." `https://www.redbooks.ibm.com/pubs/pdfs/redbooks/gg243376.pdf`.

25 "Network buffers and memory management." `https://www.linuxjournal.com/article/1312`.

26 "Anatomy of the linux networking stack." `http://140.120.7.21/LinuxRef/Network/LinuxNetworkStack.html`.

27 "Linux network stack." `https://www.linux.org/threads/linux-network-stack.9065/`.

28 "The linux tcp ip stack - networking for embedded systems." `https://www.mobt3ath.com/uplode/book/book-55476.pdf`.

29 "Socket, linux programmer's manual." `http://man7.org/linux/man-pages/man7/socket.7.html`.

30 "Linux networking and useful tips for real-time applications." `http://amsekharkernel.blogspot.com/2014/08/what-is-skb-in-linux-kernel-what-are.html`.

31 "4.8. routing tables, chapter 4. ip routing." `http://linux-ip.net/html/routing-tables.html`.

32 "Bridge(8) linux." `http://man7.org/linux/man-pages/man8/bridge.8.html`.

33 "arp(8) - linux man page." `https://linux.die.net/man/8/arp`.

34 "Network bridge." `https://wiki.archlinux.org/index.php/Network_bridge`.

35 "veth, linux programmer's manual." `http://man7.org/linux/man-pages/man4/veth.4.html`, February 2018.

36 "What is vxlan?." `https://www.juniper.net/us/en/products-services/what-is/vxlan/`, May 2019.

37 "Vxlan & linux." `https://vincent.bernat.ch/en/blog/2017-vxlan-linux`, May 2017.

38 "Virtual extensible local area network (vxlan): A framework for overlaying virtualized layer 2 networks over layer 3 networks." `https://tools.ietf.org/html/rfc7348`, May 2019.

39 "vtep, open vswitch manual." `http://man7.org/linux/man-pages/man5/vtep.5.html`, May 2019.

40 "Virtual extensible local area networking documentation." `https://www.kernel.org/doc/Documentation/networking/vxlan.txt`, May 2019.

41 "Virtual extensible lan (vxlan) overview." `https://www.arista.com/assets/data/pdf/Whitepapers/Arista_Networks_VXLAN_White_Paper.pdf`, May 2019.

42 "Design." `https://github.com/docker/libnetwork/blob/master/docs/design.md`, May 2019.

43 "Chapter 5. network setup." `https://www.debian.org/doc/manuals/debian-reference/ch05.en.html#_the_basic_network_infrastructure`.

44 "iptables (8) - linux man page." `https://linux.die.net/man/8/iptables`.

45 "linux/net/ipv4/tcp_output.c." `http://lxr.linux.no/linux+v2.6.20/net/ipv4/tcp_output.c`.

46 "Ipv4." `https://wiki.aalto.fi/display/OsProtocols/IPv4`.

47 "Chapter 10 networks." `https://www.tldp.org/LDP/tlk/net/net.html`.

48 "Chapter 17. network drivers." `https://www.oreilly.com/library/view/linux-device-drivers/0596005903/ch17.html`.

49 "Docker network drivers." `https://docs.docker.com/network/`.

50 "Use bridge networks." `https://docs.docker.com/network/bridge/`, May 2019.

51 "Loop, linux programmer's manual." `http://man7.org/linux/man-pages/man4/loop.4.html`, March 2019.

52 "bridge." `https://wiki.linuxfoundation.org/networking/bridge`, May 2019.

53 "ebtables/iptables interaction on a linux-based bridge." `http://ebtables.netfilter.org/br_fw_ia/br_fw_ia.html`, May 2019.

54 "Disable networking for a container." `https://docs.docker.com/network/none/`, May 2019.

55 "Ip forwarding." `https://docs.docker.com/network/`.

56 "Source nat." `https://www.netfilter.org/documentation/HOWTO/NAT-HOWTO-6.html`.

57 "Use overlay networks." `https://docs.docker.com/network/overlay/`, May 2019.

58 "Fast datapath & weave net." `https://www.weave.works/docs/net/latest/concepts/fastdp-how-it-works/`, May 2019.

59 "Backends." `https://github.com/coreos/flannel/blob/master/Documentation/backends.md`, May 2019.

60 "Why calico?." `https://www.projectcalico.org/why-calico/`, January 2015.

61 "Large mtus and internet performance." `http://irep.ntu.ac.uk/id/eprint/13183/1/221075_PubSub2797_Lee_K.pdf`, May 2019.

62 "ip-link (8) - linux man pages." `https://www.systutorials.com/docs/linux/man/8-ip-link/`, May 2019.

63 "Anatomy of a linux bridge." `https://wiki.aalto.fi/download/attachments/70789083/linux_bridging_review.pdf`, May 2019.

64 "An ethernet address resolution protocol." `https://tools.ietf.org/html/rfc826`, May 2019.

65 "Vxlan design and deployment." `https://www.cisco.com/c/dam/m/sl_si/events/2016/cisco_dan_inovativnih_resitev/pdf/cisco_day_slovenia_2016_vxlan_marian_klas_final.pdf`, May 2019.

66 "tc(8) - linux manual page." `https://man7.org/linux/man-pages/man8/tc.8.html`.

67 "6. classless queuing disciplines (qdiscs)." `https://tldp.org/en/Traffic-Control-HOWTO/ar01s06.html`.

68 "Queuing in the linux network stack." `https://www.coverfire.com/articles/queueing-in-the-linux-network-stack/`.

69 "The rise of kubernetes in 2017." `https://logz.io/blog/rise-kubernetes-2017/`.

70 "Concepts." `https://kubernetes.io/docs/concepts/`.

71 "Container runtimes: clarity." `https://medium.com/cri-o/container-runtimes-clarity-342b62172dc3`.

72 "Cni - the container network interface." `https://github.com/containernetworking/cni`.

73 "An introduction to kubernetes." `https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes`.

74 "kube-apiserver." `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-apiserver/`.

75 "etcd." `https://github.com/etcd-io/etcd`.

76 "Pod overview." `https://kubernetes.io/docs/concepts/workloads/pods/pod-overview/`.

77 "Service." `https://kubernetes.io/docs/concepts/services-networking/service/`.

78 "Volumes." `https://kubernetes.io/docs/concepts/storage/volumes/`.

79 "Namespaces." `https://kubernetes.io/docs/concepts/overview/working-with-objects/namespaces/`.

80 "Replicaset." `https://kubernetes.io/docs/concepts/workloads/controllers/replicaset/`.

81 "Deployments." `https://kubernetes.io/docs/concepts/workloads/controllers/deployment/`.

82 "Daemonset." `https://kubernetes.io/docs/concepts/workloads/controllers/daemonset/`.

83 "kube-controller-manager." `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-controller-manager/`.

84 "kube-scheduler." `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-scheduler/`.

85 "Introducing container runtime interface (cri) in kubernetes." `https://kubernetes.io/blog/2016/12/container-runtime-interface-cri-in-kubernetes/`.

86 "How a container runtime is using cni." `https://karampok.me/posts/container-networking-with-cni/`.

87 "Container network interface specification." `https://github.com/containernetworking/cni/blob/master/SPEC.md`.

88 "Network plugins." `https://kubernetes.io/docs/concepts/extend-kubernetes/compute-storage-net/network-plugins/`.

89 "kube-proxy." `https://kubernetes.io/docs/reference/command-line-tools-reference/kube-proxy/`.

90 "Networking overview." `https://cloud.google.com/kubernetes-engine/docs/concepts/network-overview`.

91 "Discovering services." `https://kubernetes.io/docs/concepts/services-networking/service/#discovering-services`.

92 "In search of an understandable consensus algorithm." `https://raft.github.io/raft.pdf`.

93 "swarm docs: add administration guide for managers and raft." `https://github.com/moby/moby/commit/24f87f26e73a49383e0606813a86ed96da7f5a18`.

94 "Api overview." `https://kubernetes.io/docs/reference/generated/kubernetes-api/v1.12/#nodestatus-v1-core`.

95 "Efficient node heartbeats." `https://github.com/kubernetes/enhancements/blob/master/keps/sig-node/0009-node-heartbeat.md`.

96 StackRox, "The state of container and kubernetes security."

97 "Affordable and energy-efficient cloud computing clusters: The bolzano raspberry pi cloud cluster experiment." `http://sci-hub.tw/10.1109/cloudcom.2013.121`.

98 "Vl2: A scalable and flexible data center network." `https://www.microsoft.com/en-us/research/wp-content/uploads/2016/02/vl2-sigcomm09-final.pdf`.

99 "Advanced handheld electronic banking system using raspberry pi." `http://arjst.com/pdfs/VOLUME\%204/ISSUE\%201/ARJST-04-01-053/ARJST-04-01-053\%20Advanced\%20Handheld\%20Electronic\%20banking\%20System\%20using\%20Raspberry\%20pi.pdf`.

100 "Raspbian." `https://www.raspberrypi.org/downloads/raspbian/`.

101 "About raspbian." `https://www.raspbian.org/RaspbianAbout`.

102 "Difference between tcp/ip and osi model." `https://techdifferences.com/difference-between-tcp-ip-and-osi-model.html`.

103 "kubelet." `https://kubernetes.io/docs/reference/command-line-tools-reference/kubelet/`.

104 "An introduction to kubernetes." `https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes`.

105 "Destination nat." `https://www.netfilter.org/documentation/HOWTO/NAT-HOWTO-6.html`.

106 E. Li, Zheng, M. Kihl, Q. Lu, and J. Andersson, "Performance overhead comparison between hypervisor and container based virtualization," pp. 955–962, 03 2017.

107 K. Suo, Y. Zhao, W. Chen, and J. Rao, "An analysis and empirical study of container networks," in *IEEE INFOCOM 2018 - IEEE Conference on Computer Communications*, pp. 189–197, April 2018.

108 T. C. Rajkumar Buyya, Hai Jin, "Cluster computing," Future Generation Computer Systems, 2002.

109 M. Hausenblas, *Container Networking From Docker to Kubernetes*. 1005 Gravenstein Highway North, Sebastopol, CA 95472: O'Reilly Media, 1st ed., May 2018.

110 J. M. Murugiah Souppaya and K. Scarfone, *Application Container Security Guide*. 100 Bureau Drive (Mail Stop 8930) Gaithersburg, MD 20899-8930: National Institute of Standards and Technology, special publication 800-190 ed., September 2017.

111 "Container networking." `https://docs.docker.com/config/containers/container-networking/`, May 2019.

112 "Weaveworks documentation." `https://www.weave.works/docs/`, May 2019.

113 "flannel." `https://github.com/coreos/flannel`, May 2019.

114 "About calico." `https://docs.projectcalico.org/v3.7/introduction/`, May 2019.

115 A. Ellis, "Kubernetes on (vanilla) raspbian lite." `https://github.com/teamserverless/k8s-on-raspbian/blob/master/GUIDE.md`, May 2019.

116 S. Hemminger, "Bridge, linux." `http://man7.org/linux/man-pages/man8/bridge.8.html`, August 2012.

117 "The osi model: Understanding the seven layers of computer networks." `http://ru6.cti.gr/bouras-old/WP_Simoneau_OSIModel.pdf`, May 2019.

118 "Protocols and layers." `http://www.exa.unicen.edu.ar/catedras/comdat1/material/TP1-Ejercicio5-ingles.pdf`, May 2019.

A   Custom container with a user namespace

The following script creates a process with a User Namespace to demonstrate how it can be used with containers.

```go
# container.go

package main

import (
  "fmt"
  "os"
  "os/exec"
  "syscall"
)

func main() {
  switch os.Args[1] {
    case "run":
      run()
    case "child":
      child()
    default:
      panic("help")
  }
}

func run() {
  fmt.Printf("Main %v \n", os.Args[2:])

  // This is the main process.
  // It initiates a child process.
  // The child process is the "container".
  cmd := exec.Command("/proc/self/exe", append([]string{"child"},
    os.Args[2:]...)...)
  cmd.Stdin = os.Stdin
  cmd.Stdout = os.Stdout
  cmd.Stderr = os.Stderr

  cmd.SysProcAttr = &syscall.SysProcAttr{

    // Set CLONE_NEWUSER flag for the container process.
    // This will create it with a User Namespace.
    Cloneflags: syscall.CLONE_NEWUSER,
```

```go
      // The container inherits the UID and GID of the caller.
      // Map these to zero (root) inside the container.
      UidMappings: []syscall.SysProcIDMap{
        {
          ContainerID: 0,
          HostID: os.Getuid(),
          Size: 1,
        },
      },
      GidMappings: []syscall.SysProcIDMap{
        {
          ContainerID: 0,
          HostID: os.Getgid(),
          Size: 1,
        },
      },
  }

  must(cmd.Run())
}

func child() {
  fmt.Printf("Child %v \n", os.Args[2:])

  // This is the container process.
  // Code executed here is executed inside the container.
  cmd := exec.Command(os.Args[2], os.Args[3:]...)
  cmd.Stdin = os.Stdin
  cmd.Stdout = os.Stdout
  cmd.Stderr = os.Stderr

  must(cmd.Run())
}

func must(err error) {
  if err != nil {
    panic(err)
  }
}
```

B   Custom container with a cgroup

The following script creates a process and assigns it to a cgroup. The cgroup limits the maximum number of processes that can be run by the process to 20.

```go
# container.go

package main

import (
  "fmt"
  "os"
  "os/exec"
  "path/filepath"
  "strconv"
  "io/ioutil"
)

func main() {
  switch os.Args[1] {
    case "run":
      run()
    default:
      panic("help")
  }
}

func run() {
  fmt.Printf("Main %v \n", os.Args[2:])

  cmd := exec.Command(os.Args[2], os.Args[3:]...)
  cmd.Stdin = os.Stdin
  cmd.Stdout = os.Stdout
  cmd.Stderr = os.Stderr

  cg()

  must(cmd.Run())
}

// Create a cgroup and assign this process to it
func cg() {

  // The mount location of the cgroup filesystem
  cgroups := "/sys/fs/cgroup/"
```

```go
	// Choose the resource controller pids
	pids := filepath.Join(cgroups, "pids")

	// Create a cgroup called container
	os.Mkdir(filepath.Join(pids, "container"), 0755)

	// Add pids.max to the cgroup
	// Limit maximum number of processes to 20
	must(ioutil.WriteFile(filepath.Join(pids, "container/pids.max"),
		[]byte("20"), 0700))

	// Remove the  cgroup after the container exits
	must(ioutil.WriteFile(filepath.Join(pids,
		"container/notify_on_release"), []byte("1"), 0700))

	// Write the container PID to the file cgroup.procs
	// Now the container will belong to the new cgroup
	must(ioutil.WriteFile(filepath.Join(pids, "container/cgroup.procs"),
		[]byte(strconv.Itoa(os.Getpid())), 0700))
}

func must(err error) {
	if err != nil {
		panic(err)
	}
}
```