Community Experience Distilled

Learning Docker Networking

Become a proficient Linux administrator by learning the art of container networking with elevated efficiency using Docker

Rajdeep Dua
Vaibhav Kohli
Santosh Kumar Konduri

[PACKT] PUBLISHING open source*
community experience distilled

www.allitebooks.com

www.allitebooks.com

Learning Docker Networking

www.allitebooks.com

Table of Contents

Learning Docker Networking

Credits

About the Authors

About the Reviewer

www.PacktPub.com

- eBooks, discount offers, and more
  - Why subscribe?

Preface

- What this book covers
- What you need for this book
- Who this book is for
- Conventions
- Reader feedback
- Customer support
  - Downloading the example code
  - Downloading the color images of this book
  - Errata
  - Piracy
  - Questions

1. Docker Networking Primer

- Networking and Docker
  - Linux bridges
  - Open vSwitch
  - NAT
  - IPtables
  - AppArmor/SELinux
- The docker0 bridge
  - The —net default mode
  - The —net=none mode

www.allitebooks.com

    - The —net=container:$container2 mode
    - The —net=host mode
      - Port mapping in Docker container
  - Docker OVS
  - Unix domain socket
  - Linking Docker containers
    - Links
  - What’s new in Docker networking?
    - Sandbox
    - Endpoint
    - Network
  - The Docker CNM model
  - Summary
- 2. Docker Networking Internals
  - Configuring the IP stack for Docker
    - IPv4 support
    - IPv6 support
  - Configuring a DNS server
    - Communication between containers and external networks
      - Restricting SSH access from one container to another
  - Configuring the Docker bridge
  - Overlay networks and underlay networks
  - Summary
- 3. Building Your First Docker Network
  - Introduction to Pipework
  - Multiple containers over a single host
    - Weave your containers
  - Open vSwitch
    - Single host OVS
      - Creating an OVS bridge
    - Multiple host OVS

www.allitebooks.com

- Networking with overlay networks – Flannel
- Summary

4. Networking in a Docker Cluster
  - Docker Swarm
    - Docker Swarm setup
    - Docker Swarm networking
  - Kubernetes
    - Deploying Kubernetes on AWS
    - Kubernetes networking and its differences to Docker networking
    - Deploying the Kubernetes pod
  - Mesosphere
    - Docker containers
    - Deploying a web app using Docker
    - Deploying Mesos on AWS using DCOS
  - Summary

5. Security and QoS for Docker Containers
  - Filesystem restrictions
    - Read-only mount points
      - sysfs
      - procfs
      - /dev/pts
      - /sys/fs/cgroup
    - Copy-on-write
  - Linux capabilities
  - Securing containers in AWS ECS
  - Understanding Docker security I – kernel namespaces
    - pid namespace
    - net namespace
      - Basic network namespace management
      - Network namespace configuration
    - User namespace

www.allitebooks.com

      - Creating a new user namespace
  - Understanding Docker security II – cgroups
    - Defining cgroups
    - Why are cgroups required?
    - Creating a cgroup manually
    - Attaching processes to cgroups
    - Docker and cgroups
  - Using AppArmor to secure Docker containers
    - AppArmor and Docker
    - Docker security benchmark
      - Audit Docker daemon regularly
      - Create a user for the container
      - Do not mount sensitive host system directories on containers
      - Do not use privileged containers
  - Summary
- 6. Next Generation Networking Stack for Docker: libnetwork
  - Goal
  - Design
  - CNM objects
    - Sandbox
    - Endpoint
    - Network
    - Network controller
    - CNM attributes
    - CNM lifecycle
  - Driver
    - Bridge driver
    - Overlay network driver
  - Using overlay network with Vagrant
    - Overlay network deployment Vagrant setup
  - Overlay network with Docker Machine and Docker Swarm

www.allitebooks.com

    - Prerequisites
    - Key-value store installation
    - Create a Swarm cluster with two nodes
    - Creating an overlay network
  - Creating containers using an overlay network
    - Container network interface
  - CNI plugin
    - Network configuration
    - IP allocation
    - IP address management interface
  - Project Calico’s libnetwork driver
  - Summary
- Index

www.allitebooks.com

www.allitebooks.com

Learning Docker Networking

www.allitebooks.com

Learning Docker Networking

Copyright © 2016 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author(s), nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: February 2016

Production reference: 1190216

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78528-095-5

www.packtpub.com

Credits

Authors

Rajdeep Dua

Vaibhav Kohli

Santosh Kumar Konduri

Reviewer

Jon Langemak

Commissioning Editor

Kunal Parikh

Acquisition Editor

Tushar Gupta

Content Development Editor

Mayur Pawanikar

Technical Editor

Dhiraj Chandanshive

Copy Editors

Joanna McMahon

Madhusudan Uchil

Project Coordinator

Nidhi Joshi

Proofreader

Safis Editing

Indexer

Rekha Nair

Graphics

Jason Monteiro

Production Coordinator

Aparna Bhagat

Cover Work

Aparna Bhagat

About the Authors

Rajdeep Dua has over 16 years of experience in distributed systems. He has worked in R&D and Developer Relation roles at Microsoft, Google, VMware, and Salesforce.com. He has exposure to multiple cloud platforms like Google App Engine, Heroku, Force.com, vSphere, and Google Compute Engine.

Rajdeep has been working on Docker and related container technologies for more than two years now. He did his MBA in IT from IIM Lucknow in the year 2000.

Vaibhav Kohli has around 3 years of working experience in the research and development department of VMware, and he has been teaching computer engineering for a year at the esteemed Mumbai University. He has published many research papers and filed three patents from VMware in the container domain. He has also conducted workshops in various companies and meetups on container technology (Docker) and Kubernetes.

Santosh Kumar Konduri has around 5 years of IT experience. He is an expert OpenStack administrator with 3 years of experience.

About the Reviewer

Jon Langemak has over 10 years of experience in designing, building, and maintaining high-performance networks. He’s currently employed as a network architect at a Minnesota-based company, where he focuses on disruptive technologies and the impact they have on network operations. Outside of work, Jon blogs at www.dasblinkenlichten.com and enjoys collaborating with others in the networking community on new ideas and concepts.

www.PacktPub.com

eBooks, discount offers, and more

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <`customercare@packtpub.com`> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt’s online digital book library. Here, you can search, access, and read Packt’s entire library of books.

Why subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print, and bookmark content
- On demand and accessible via a web browser

Preface

This book helps the reader to learn, create, deploy, and provide administration steps for Docker networking. Docker is a Linux container implementation that enables the creation of light-weight portable development and production-quality environments. These environments can be updated incrementally. Docker achieves this by leveraging containment principles, such as cgroups and Linux namespaces, along with overlay filesystem-based portable images.

Docker provides the networking primitives that allow administrators to specify how different containers network with each application, connect to each of their components, then distribute them across a large number of servers, and ensure coordination between them irrespective of the host or the VM that they are running on. This book aggregates all the latest Docker networking technology and provides great in depth explanation with setup details.

What this book covers

Chapter 1, *Docker Networking Primer*, explains the essential components of Docker networking, which have evolved from coupling simple Docker abstractions and powerful network components, such as Linux bridges, Open vSwitch, and so on. This chapter also explains how Docker containers can be created with various modes. In the default mode, port mapping helps us through the use of iptables NAT rules, allowing traffic arriving at the host to reach containers. Later in this chapter, basic linking of the container is covered and the next generation of Docker networking, which is libnetwork, is also discussed.

Chapter 2, *Docker Networking Internals*, discusses Docker's internal networking architecture. We will learn about IPv4, IPv6, and DNS configurations in Docker. Later in this chapter, Docker bridge and communication between containers in single host and multihost is covered. This chapter also explains overlay tunneling and different methods that are implemented on Docker networking, such as OVS, Flannel, and Weave.

Chapter 3, *Building Your First Docker Network*, shows how Docker containers communicate from multiple hosts using different networking options, such as Weave, OVS, and Flannel. Pipework uses legacy Linux bridge, Weave creates a virtual network, OVS uses GRE tunneling technology, and Flannel provides a separate subnet to each host to connect containers on multiple hosts. Some of the implementations, such as Pipework, are legacy and will become obsolete over a period of time, while others are designed to be used in the context of specific OSes, such as Flannel with CoreOS. Basic comparisons of Docker networking options are also covered in this chapter.

Chapter 4, *Networking in a Docker Cluster*, explains Docker networking in depth using various frameworks, such as native Docker Swarm, where using the libnetwork or the out-of the-box overlay network, Swarm, provides the multihost networking features. Kubernetes, on the other hand, has a different perspective from Docker, where each pod will get a unique IP address and communication between pods can occur with the help of services. Using Open vSwitch or IP-forwarding advanced routing rules, the Kubernetes networking can be enhanced to provide connectivity between pods on different subnets across hosts and expose the pods to the external world. In the case of Mesosphere, we can see that Marathon is used as a backend for the networking of the deployed containers. In the case of DCOS of the Mesosphere, the entire deployed stack of machines is treated as one machine to provide a rich-networking experience between deployed container services.

Chapter 5, *Security and QoS for Docker Containers*, takes a dive into Docker security by referring to kernel and cgroups namespaces. We will also visit some of the aspects of filesystems and various Linux capabilities, which containers leverage in order to provide more features, such as the privileged container but at the cost of exposing itself more on the threat side. We will also see how containers can be deployed in a secured environment in AWS ECS using proxy containers to restrict the vulnerable traffic. We will also talk about how AppArmor is also provided with a rich set of **Mandatory Access Control** (**MAC**) system, which provides kernel-enhancement features in order to confine applications to a limited set of resources. Leveraging their benefits to Docker containers helps us deploy them in a secured environment. In the last section, we take a quick dive into Docker security benchmarks and some of the important recommendations that can be followed during auditing and Docker deployment in a production environment.

Chapter 6, *Next Generation Networking Stack for Docker: libnetwork*, will look into some of the deeper and conceptual aspects of Docker networking. One of these is libnetworking—the future of the Docker network model, which is already getting into shape with the release of Docker 1.9. While explaining the libnetworking concept, we will also study the CNM model, its various objects and components, along with its implementation code snippets. Next, we will look into drivers of CNM, the prime one being the overlay driver, in detail with deployment as part of Vagrant setup. We will look at standalone integrations of containers with overlay network with Docker Swarm and Docker Machine as well. In the next section, we explain the CNI interface, its executable plugins, and give a tutorial to configure Docker networking with the CNI plugin. In the last section, Project Calico is explained in detail, which provides scalable networking solutions that are based out of libnetwork and provides integration with Docker, Kubernetes, Mesos, bare-metal, and VMs, primarily.

What you need for this book

Mostly all of the setups basically require Ubuntu 14.04 (either installed on a physical machine or as a virtual machine) and Docker 1.9, which is the latest version to date. Specific OS and software requirements (Open Source Git Projects) are mentioned before each setup if required.

Who this book is for

If you are a Linux administrator who wants to learn networking using Docker to ensure the efficient administration of core elements and applications, then this book is for you. Basic knowledge of LXC/Docker is assumed.

www.allitebooks.com

Conventions

You will also find a number of text styles that distinguish between different kinds of information. Here are some examples of these styles and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "You may notice that we used the Unix command, `rm`, to remove the `Drush` directory rather than the DOS `del` command."

A block of code is set as follows:

```
# * Fine Tuning
#
key_buffer = 16M
key_buffer_size = 32M
max_allowed_packet = 16M
thread_stack = 512K
thread_cache_size = 8
max_connections = 300
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
# * Fine Tuning
#
key_buffer = 16M
key_buffer_size = 32M
max_allowed_packet = 16M
thread_stack = 512K
thread_cache_size = 8
max_connections = 300
```

Any command-line input or output is written as follows:

```
cd /ProgramData/Propeople
rm -r Drush
git clone --branch master http://git.drupal.org/project/drush.git
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "On the **Select Destination Location** screen, click on **Next** to accept the default destination."

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or disliked. Reader feedback is important for us as it helps us develop titles that you will really get the most out of.

To send us general feedback, simply e-mail <`feedback@packtpub.com`>, and mention the book's title in the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide at www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for this book from your account at http://www.packtpub.com. If you purchased this book elsewhere, you can visit http://www.packtpub.com/support and register to have the files e-mailed directly to you.

You can download the code files by following these steps:

1. Log in or register to our website using your e-mail address and password.
2. Hover the mouse pointer on the **SUPPORT** tab at the top.
3. Click on **Code Downloads & Errata**.
4. Enter the name of the book in the **Search** box.
5. Select the book for which you're looking to download the code files.
6. Choose from the drop-down menu where you purchased this book from.
7. Click on **Code Download**.

Once the file is downloaded, please make sure that you unzip or extract the folder using the latest version of:

- WinRAR / 7-Zip for Windows
- Zipeg / iZip / UnRarX for Mac
- 7-Zip / PeaZip for Linux

Downloading the color images of this book

We also provide you with a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from http://www.packtpub.com/sites/default/files/downloads/LearningDockerNetworking_Color

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you could report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/submit-errata, selecting your book, clicking on the **Errata Submission Form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works in any form on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <`copyright@packtpub.com`> with a link to the suspected pirated material.

We appreciate your help in protecting our authors and our ability to bring you valuable content.

Questions

If you have a problem with any aspect of this book, you can contact us at <`questions@packtpub.com`>, and we will do our best to address the problem.

Chapter 1. Docker Networking Primer

Docker is a lightweight container technology that has gathered enormous interest in recent years. It neatly bundles various Linux kernel features and services, such as namespaces, cgroups, SELinux, and AppArmor profiles, over union filesystems such as AUFS and BTRFS in order to make modular images. These images provide a highly configurable virtualized environment for applications and follow a **write once, run anywhere** workflow. An application can be composed of a single process running in a Dcker container or it could be made up of multiple processes running in their own containers and being replicated as the load increases. Therefore, there is a need for powerful networking elements that can support various complex use cases.

In this chapter, you will learn about the essential components of Docker networking and how to build and run simple container examples.

This chapter covers the following topics:

- Networking and Docker
- The `docker0` bridge networking
- Docker OVS networking
- Unix domain networks
- Linking Docker containers
- What's new in Docker networking

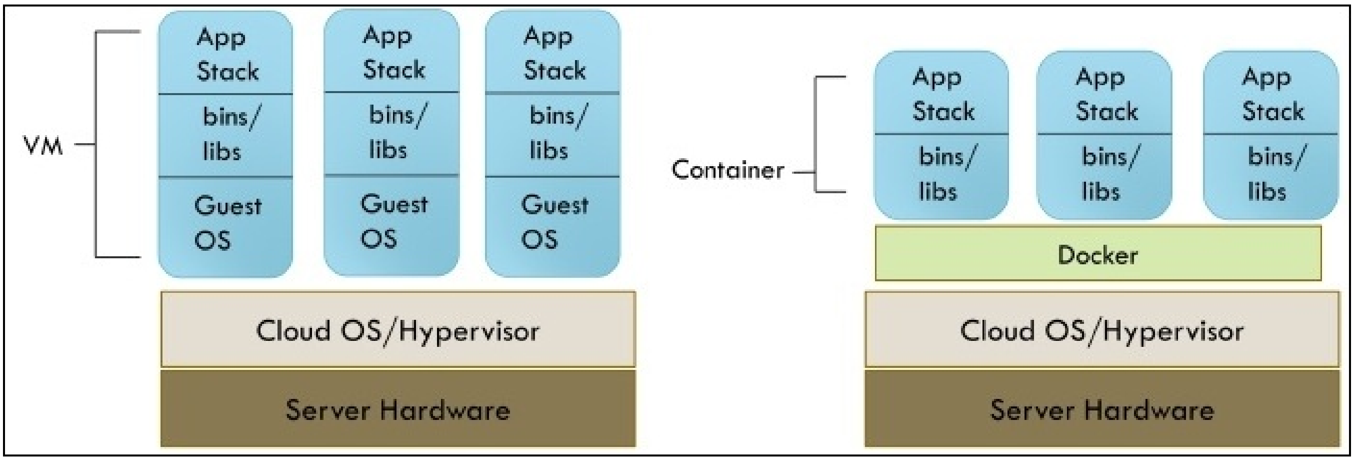
Docker is getting a lot of traction in the industry because of its performance-savvy and universal replicability architecture, while providing the following four cornerstones of modern application development:

- Autonomy
- Decentralization
- Parallelism
- Isolation

~~Furthermore, wide-scale adoption of Thoughtworks's microservices architecture, or **LOSA** (**Lots of Small Applications**), is further bringing potential to Docker technology. As a result, big companies such as Google, VMware, and Microsoft have already ported Docker to their infrastructure, and the momentum is continued by the launch of myriad Docker start-ups, namely Tutum, Flocker, Giantswarm, and so on~~.

Since Docker containers replicate their behavior anywhere, be it your development machine, a bare metal server, virtual machine, or data center, application designers can focus their attention on development, while operational semantics are left with DevOps. This makes team workflow modular, efficient, and productive. Docker is not to be confused with a **virtual machine** (**VM**), even though they are both virtualization technologies. While Docker shares an OS with providing a sufficient level of isolation and security to applications running in containers, it later completely abstracts away the OS and gives strong isolation and security guarantees. However, Docker's resource footprint is minuscule in comparison to a VM and hence preferred for economy and performance.

However, it still cannot completely replace VMs and is therefore complementary to VM technology. The following diagram shows the architecture of VMs and Docker:


VM
App Stack
bins/libs
Guest OS
App Stack
bins/libs
Guest OS
App Stack
bins/libs
Guest OS
Cloud OS/Hypervisor
Server Hardware
Container
App Stack
bins/libs
App Stack
bins/libs
App Stack
bins/libs
Docker
Cloud OS/Hypervisor
Server Hardware

Networking and Docker

Each Docker container has its own network stack, and this is due to the Linux kernel NET namespace, where a new NET namespace for each container is instantiated and cannot be seen from outside the container or from other containers.

Docker networking is powered by the following network components and services.

Linux bridges

These are L2/MAC learning switches built into the kernel and are to be used for forwarding.

Open vSwitch

This is an advanced bridge that is programmable and supports tunneling.

NAT

Network address translators are immediate entities that translate IP addresses and ports (SNAT, DNAT, and so on).

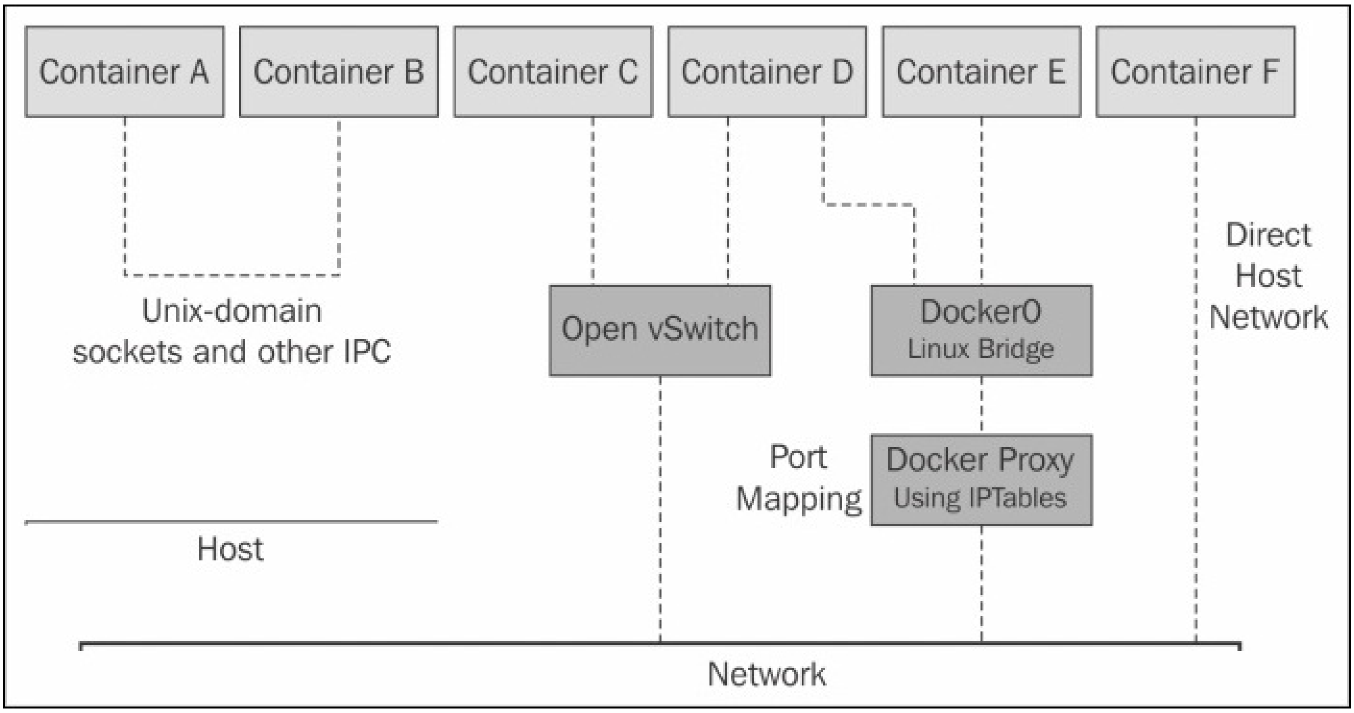
IPtables

This is a policy engine in the kernel used for managing packet forwarding, firewall, and NAT features.

AppArmor/SELinux

Firewall policies for each application can be defined with these.

Various networking components can be used to work with Docker, providing new ways to access and use Docker-based services. As a result, we see a lot of libraries that follow a different approach to networking. Some of the prominent ones are Docker Compose, Weave, Kubernetes, Pipework, libnetwork, and so on. The following figure depicts the root ideas of Docker networking:


Container A
Container B
Container C
Container D
Container E
Container F
Direct Host Network
Unix-domain sockets and other IPC
Open vSwitch
Docker0 Linux Bridge
Port Mapping
Docker Proxy Using IPTables
Host
Network

www.allitebooks.com

The docker0 bridge

The `docker0` bridge is the heart of default networking. When the Docker service is started, a Linux bridge is created on the host machine. The interfaces on the containers talk to the bridge, and the bridge proxies to the external world. Multiple containers on the same host can talk to each other through the Linux bridge.

`docker0` can be configured via the `--net` flag and has, in general, four modes:

- `--net default`
- `--net=none`
- `--net=container:$container2`
- `--net=host`

The —net default mode

In this mode, the default bridge is used as the bridge for containers to connect to each other.

The —net=none mode

With this mode, the container created is truly isolated and cannot connect to the network.

The —net=container:$container2 mode

With this flag, the container created shares its network namespace with the container called `$container2`.

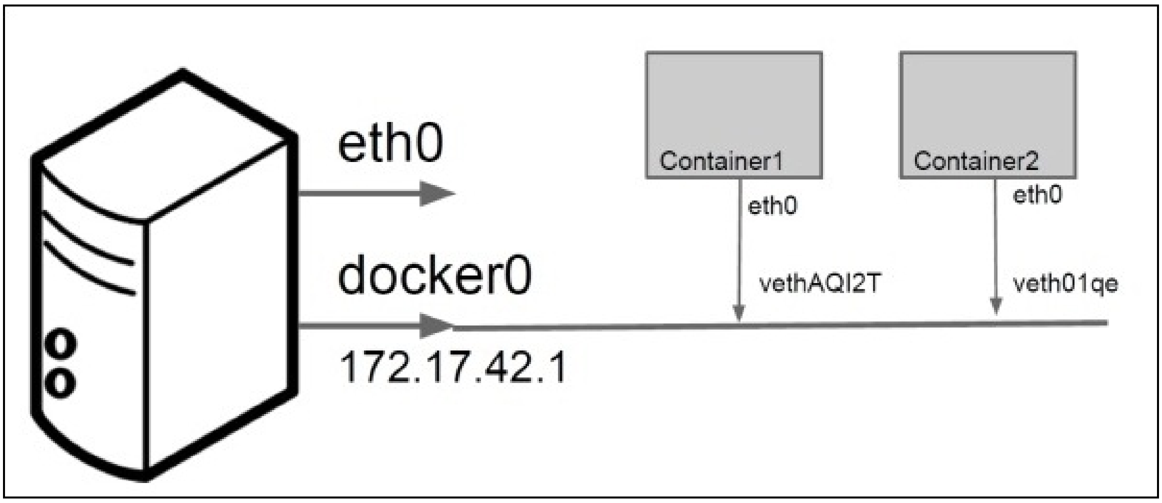
The —net=host mode

With this mode, the container created shares its network namespace with the host.

Port mapping in Docker container

In this section, we look at how container ports are mapped to host ports. This mapping can either be done implicitly by Docker Engine or can be specified.

If we create two containers called **Container1** and **Container2**, both of them are assigned an IP address from a private IP address space and also connected to the **docker0** bridge, as shown in the following figure:



Both the preceding containers will be able to ping each other as well as reach the external world.

For external access, their port will be mapped to a host port.

As mentioned in the previous section, containers use network namespaces. When the first container is created, a new network namespace is created for the container. A vEthernet link is created between the container and the Linux bridge. Traffic sent from `eth0` of the container reaches the bridge through the vEthernet interface and gets switched thereafter. The following code can be used to show a list of Linux bridges:

```
# show linux bridges
$ sudo brctl show
```

The output will be similar to the one shown as follows, with a bridge name and the `veth` interfaces on the containers it is mapped to:

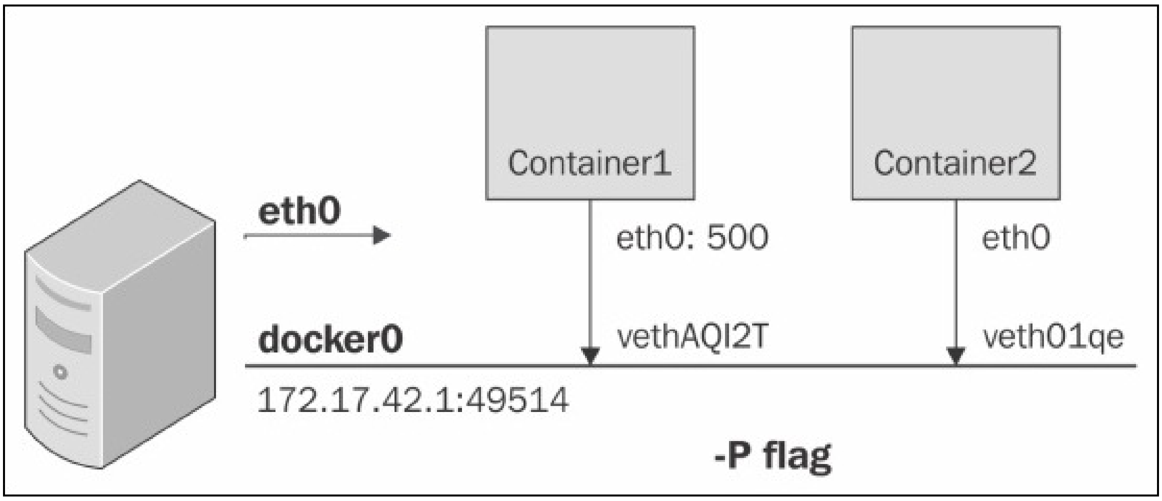
```
bridge name      bridge id        STP enabled    interfaces
docker0      8000.56847afe9799        no         veth44cb727
                                                 veth98c3700
```

How does the container connect to the external world? The `iptables nat` table on the host is used to masquerade all external connections, as shown here:

```
$ sudo iptables -t nat -L –n
```

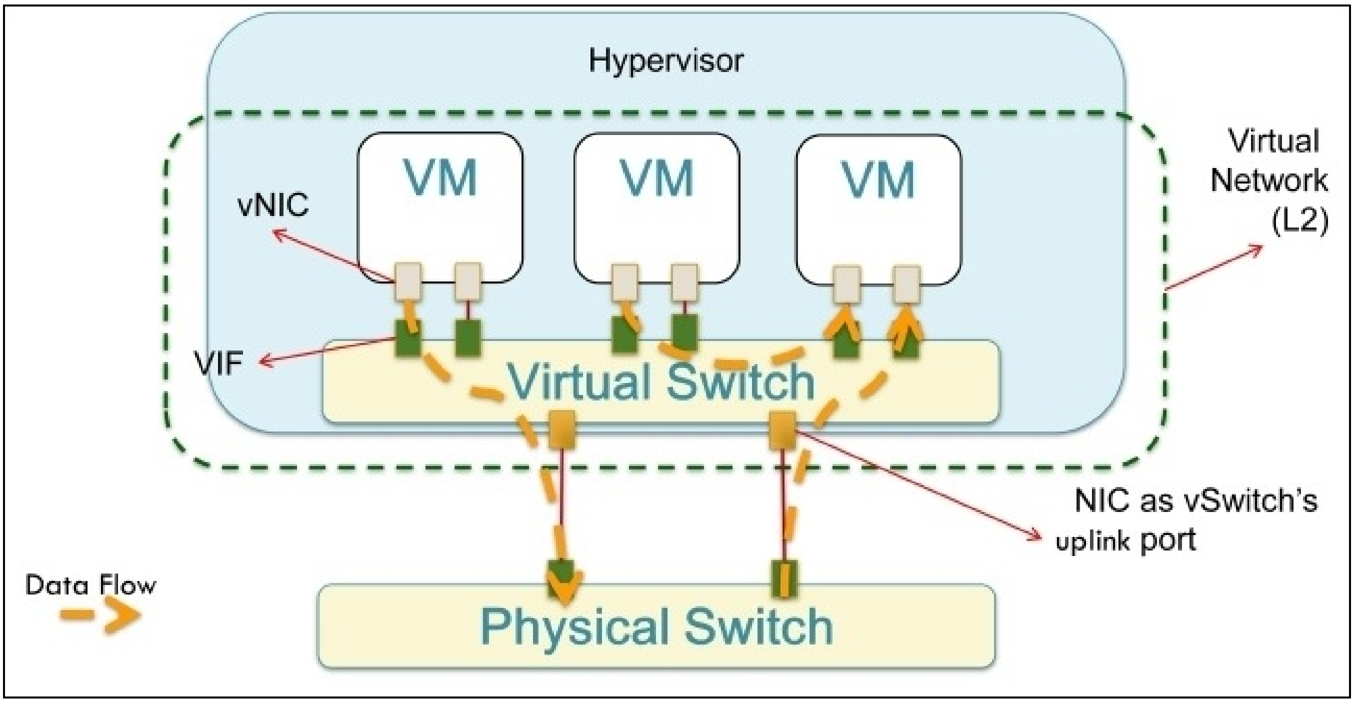
```
...
Chain POSTROUTING (policy ACCEPT) target prot opt
source destination MASQUERADE all—172.17.0.0/16
!172.17.0.0/16
...
```

How to reach containers from the outside world? The port mapping is again done using the `iptables nat` option on the host machine.

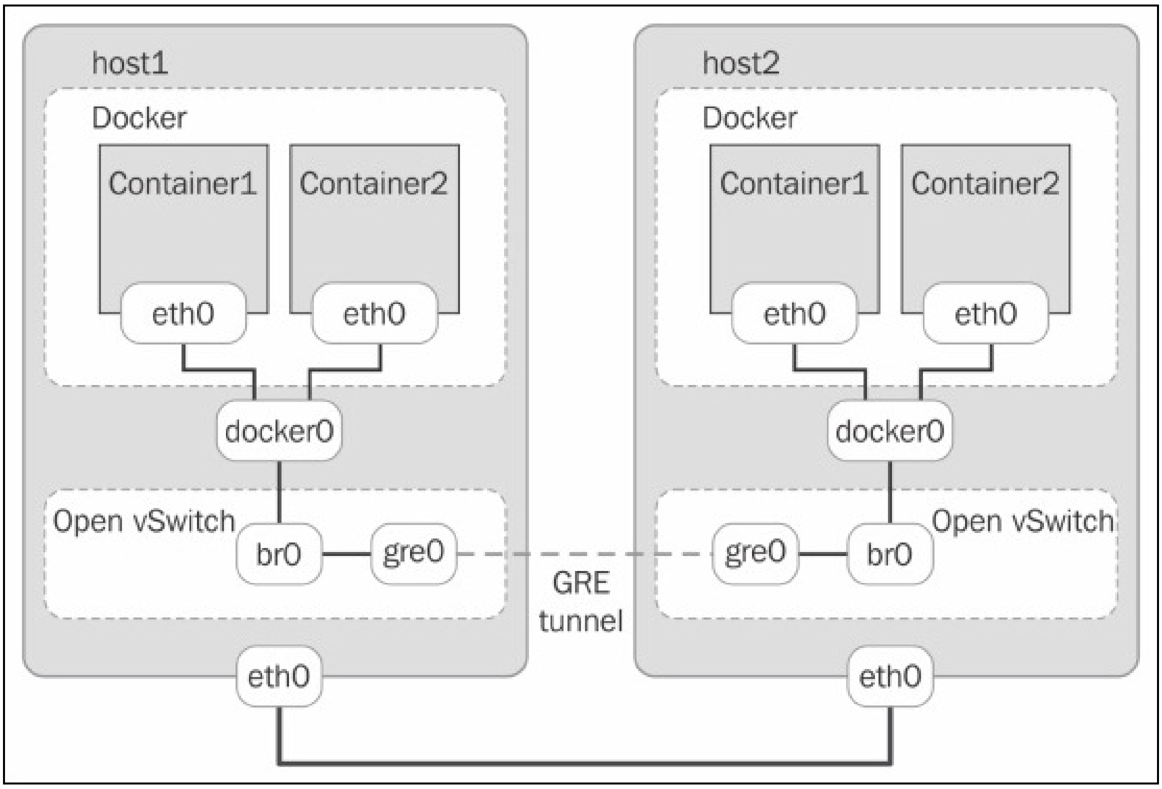

Container1
Container2
eth0
eth0: 500
eth0
docker0
vethAQI2T
veth01qe
172.17.42.1:49514
-P flag

Docker OVS

Open vSwitch is a powerful network abstraction. The following figure shows how OVS interacts with the **VM**s, **Hypervisor**, and the **Physical Switch**. Every **VM** has a **vNIC** associated with it. Every **vNIC** is connected through a **VIF** (also called a **virtual interface**) with the **Virtual Switch**:


Hypervisor
Virtual Network (L2)
VM
VM
VM
vNIC
VIF
Virtual Switch
NIC as vSwitch's uplink port
Data Flow
Physical Switch

OVS uses tunnelling mechanisms such as GRE, VXLAN, or STT to create virtual overlays instead of using physical networking topologies and Ethernet components. The following figure shows how OVS can be configured for the containers to communicate between multiple hosts using GRE tunnels:


host1
Docker
Container1
Container2
eth0
eth0
docker0
Open vSwitch
br0
gre0
eth0
host2
Docker
Container1
Container2
eth0
eth0
docker0
Open vSwitch
gre0
br0
eth0
GRE
tunnel

Unix domain socket

Within a single host, UNIX IPC mechanisms, especially UNIX domain sockets or pipes, can also be used to communicate between containers:

```
$  docker run  --name c1 –v /var/run/foo:/var/run/foo –d –I –t base /bin/bash
$  docker run  --name c2 –v /var/run/foo:/var/run/foo –d –I –t base /bin/bash
```

Apps on `c1` and `c2` can communicate over the following Unix socket address:

```
struct  sockaddr_un address;
address.sun_family = AF_UNIX;
snprintf(address.sun_path, UNIX_PATH_MAX, "/var/run/foo/bar" );
```

C1: Server.c	**C2: Client.c**

`bind(socket_fd, (struct sockaddr *) &address, sizeof(struct sockaddr_un));`
`listen(socket_fd, 5);`
`while((connection_fd = accept(socket_fd, (struct sockaddr *) &address, &address_length)) > -1)`
`nbytes = read(connection_fd, buffer, 256);`
	`connect(socket_fd, (struct sockaddr *) &address, sizeof(struct sockaddr_un));`
	`write(socket_fd, buffer, nbytes);`

Linking Docker containers

In this section, we introduce the concept of linking two containers. Docker creates a tunnel between the containers, which doesn't need to expose any ports externally on the container. It uses environment variables as one of the mechanisms for passing information from the parent container to the child container.

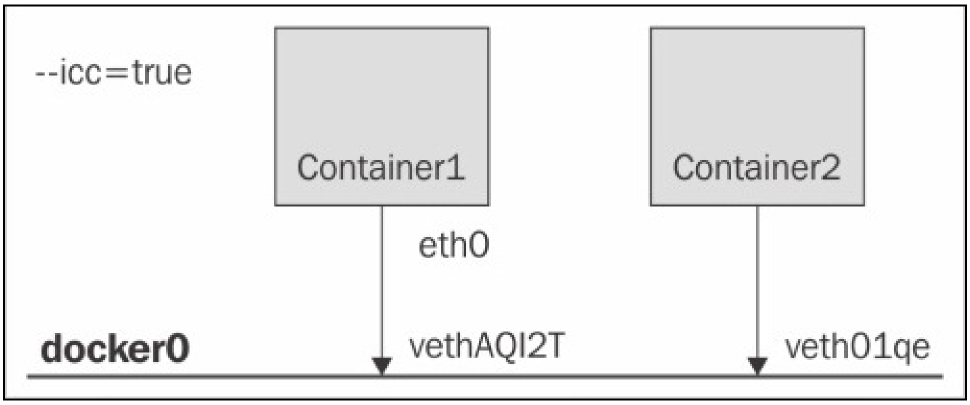
In addition to the environment variable `env`, Docker also adds a host entry for the source container to the `/etc/hosts` file. The following is an example of the host file:

```
$ docker run -t -i --name c2 --rm --link c1:c1alias training/webapp
/bin/bash
root@<container_id>:/opt/webapp# cat /etc/hosts
172.17.0.1  aed84ee21bde
...
172.17.0.2  c1alaias 6e5cdeb2d300 c1
```

There are two entries:

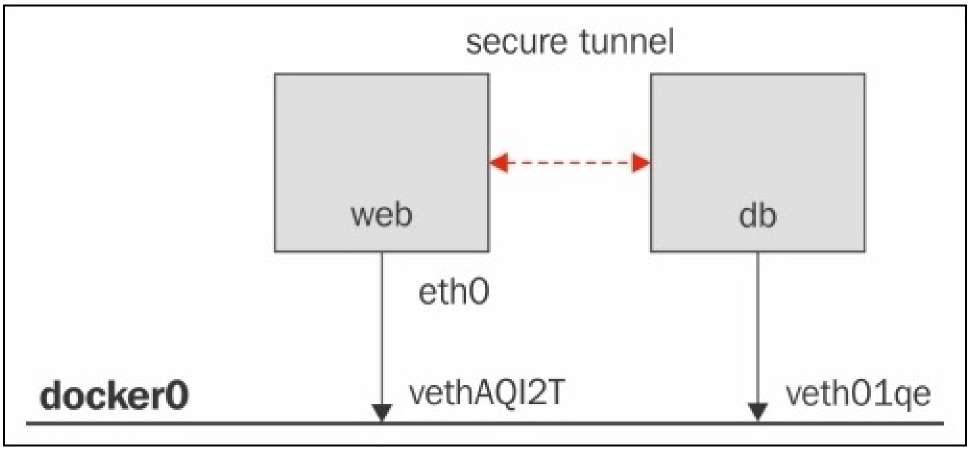
- The first is an entry for the container `c2` that uses the Docker container ID as a host name
- The second entry, `172.17.0.2 c1alaias 6e5cdeb2d300 c1`, uses the `link` alias to reference the IP address of the `c1` container

The following figure shows two containers **Container 1** and **Container 2** connected using veth pairs to the `docker0` bridge with `--icc=true`. This means these two containers can access each other through the bridge:



Links

Links provide service discovery for Docker. They allow containers to discover and securely communicate with each other by using the flag `-link name:alias`. Inter-container communication can be disabled with the daemon flag `-icc=false`. With this flag set to `false`, **Container 1** cannot access **Container 2** unless explicitly allowed via a link. This is a huge advantage for securing your containers. When two containers are linked together, Docker creates a parent-child relationship between them, as shown in the following figure:



From the outside, it looks like this:

```
# start the database
$  sudo docker run -dp 3306:3306 --name todomvcdb \
-v /data/mysql:/var/lib/mysql cpswan/todomvc.mysql

# start the app server
$  sudo docker run -dp 4567:4567 --name todomvcapp \
--link todomvcdb:db cpswan/todomvc.sinatra
```

On the inside, it looks like this:

```
$  dburl = ''mysql://root:pa55Word@'' + \ ENV[''DB_PORT_3306_TCP_ADDR''] +
''/todomvc''
$  DataMapper.setup(:default, dburl)
```

What’s new in Docker networking?

Docker networking is at a very nascent stage, and there are many interesting contributions from the developer community, such as Pipework, Weave, Clocker, and Kubernetes. Each of them reflects a different aspect of Docker networking. We will learn about them in later chapters. Docker, Inc. has also established a new project where networking will be standardized. It is called **libnetwork**.

libnetwork implements the **container network model** (**CNM**), which formalizes the steps required to provide networking for containers while providing an abstraction that can be used to support multiple network drivers. The CNM is built on three main components—sandbox, endpoint, and network.

Sandbox

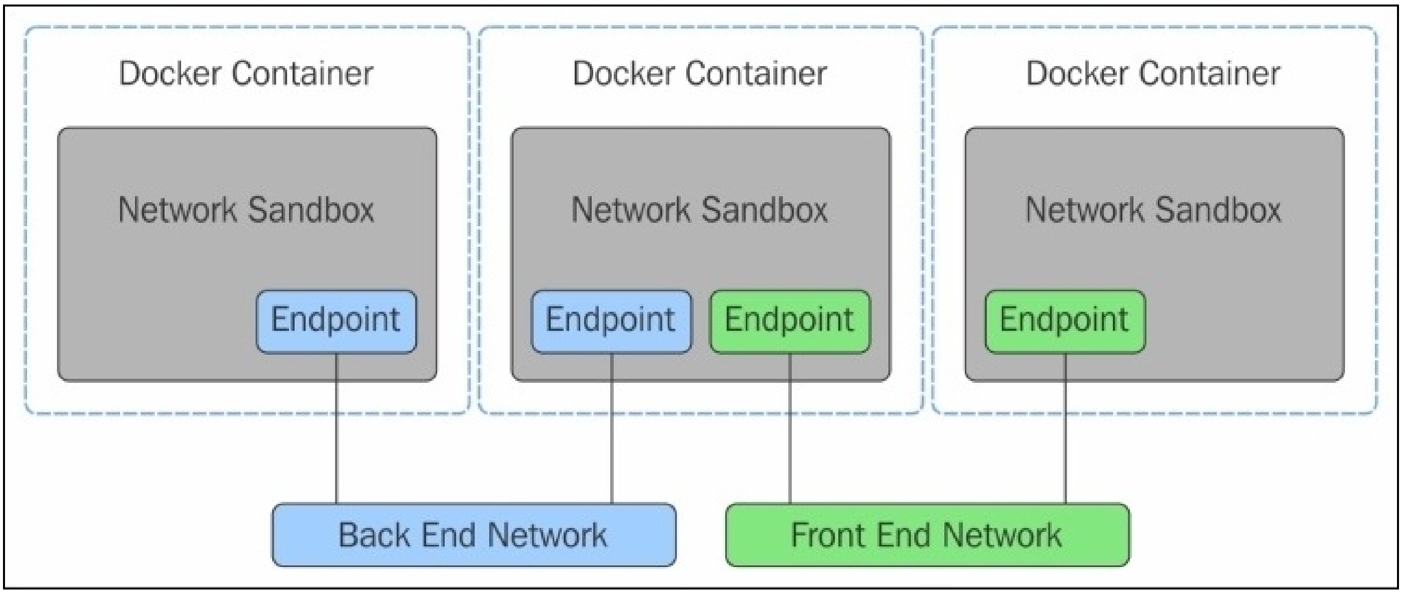
A sandbox contains the configuration of a container’s network stack. This includes management of the container’s interfaces, routing table, and DNS settings. An implementation of a sandbox could be a Linux network namespace, a FreeBSD jail, or other similar concept. A sandbox may contain many endpoints from multiple networks.

Endpoint

An endpoint connects a sandbox to a network. An implementation of an endpoint could be a veth pair, an Open vSwitch internal port, or something similar. An endpoint can belong to only one network but may only belong to one sandbox.

Network

A network is a group of endpoints that are able to communicate with each other directly. An implementation of a network could be a Linux bridge, a VLAN, and so on. Networks consist of many endpoints, as shown in the following diagram:


Docker Container
Docker Container
Docker Container
Network Sandbox
Network Sandbox
Network Sandbox
Endpoint
Endpoint
Endpoint
Endpoint
Back End Network
Front End Network

The Docker CNM model

The CNM provides the following contract between networks and containers:

- All containers on the same network can communicate freely with each other
- Multiple networks are the way to segment traffic between containers and should be supported by all drivers
- Multiple endpoints per container are the way to join a container to multiple networks
- An endpoint is added to a network sandbox to provide it with network connectivity

We will discuss the details of how CNM is implemented in Chapter 6, *Next Generation Networking Stack for Docker: libnetwork*.

Summary

In this chapter, we learned about the essential components of Docker networking, which have evolved from coupling simple Docker abstractions and powerful network components such as Linux bridges and Open vSwitch.

We learned how Docker containers can be created with various modes. In the default mode, port mapping helps through the use of iptables NAT rules, allowing traffic arriving at the host to reach containers. Later in the chapter, we covered the basic linking of containers. We also talked about the next generation of Docker networking, which is called libnetwork.

Chapter 2. Docker Networking Internals

This chapter discusses the semantics and syntax of Docker networking in detail, exposing strengths and weaknesses of the current Docker network paradigm.

It covers the following topics:

- Configuring the IP stack for Docker
  - IPv4 support
  - Issues with IPv4 address management
  - IPv6 support
- Configuring DNS
  - DNS basics
  - Multicast DNS
- Configuring the Docker bridge
- Overlay networks and underlay networks
  - What are they?
  - How does Docker use them?
  - What are some of their advantages?

Configuring the IP stack for Docker

Docker uses the IP stack to interact with the outside world using TCP or UDP. It supports the IPv4 and IPv6 addressing infrastructures, which are explained in the following subsections.

IPv4 support

By default, Docker provides IPv4 addresses to each container, which are attached to the default `docker0` bridge. The IP address range can be specified while starting the Docker daemon using the `--fixed-cidr` flag, as shown in the following code:

```
$ sudo docker –d --fixed-cidr=192.168.1.0/25
```

We will discuss more about this in the *Configuring the Docker bridge* section.

The Docker daemon can be listed on an IPv4 TCP endpoint in addition to a Unix socket:

```
$ sudo docker -H tcp://127.0.0.1:2375 -H unix:///var/run/docker.sock -d &
```

IPv6 support

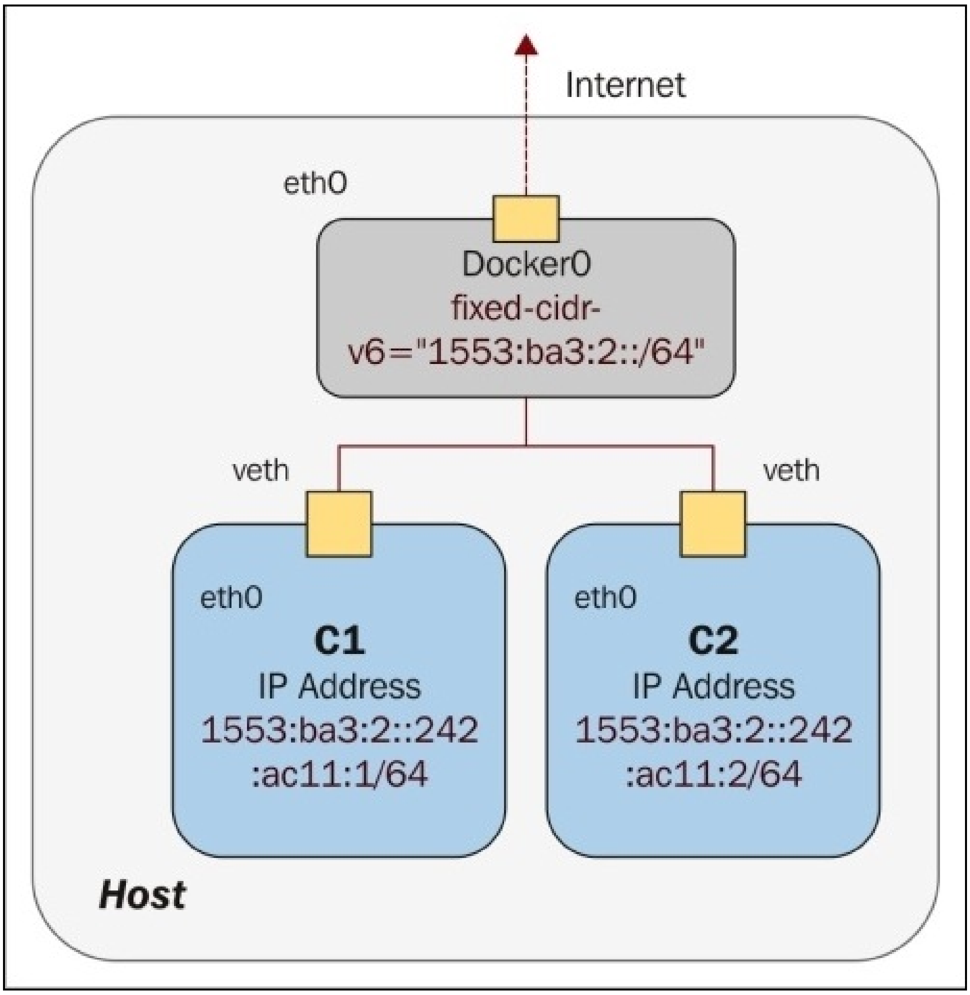
IPv4 and IPv6 can run together; this is called a **dual stack**. This dual stack support is enabled by running the Docker daemon with the `--ipv6` flag. Docker will set up the `docker0` bridge with the IPv6 link-local address `fe80::1`. All packets shared between containers flow through this bridge.

To assign globally routable IPv6 addresses to your containers, you have to specify an IPv6 subnet to pick the addresses from.

The following commands set the IPv6 subnet via the `--fixed-cidr-v6` parameter while starting Docker and also add a new route to the routing table:

```
# docker –d --ipv6 --fixed-cidr-v6="1553:ba3:2::/64"
# docker run -t -i --name c0 ubuntu:latest /bin/bash
```

The following figure shows a Docker bridge configured with an IPv6 address range:


Internet
eth0
Docker0
fixed-cidr-v6="1553:ba3:2::/64"
veth
veth
eth0
C1
IP Address
1553:ba3:2::242:ac11:1/64
eth0
C2
IP Address
1553:ba3:2::242:ac11:2/64
Host

If you check the IP address range using `ifconfig` inside a container, you will notice that the appropriate subnet has been assigned to the `eth0` interface, as shown in the following code:

```
#ifconfig
eth0      Link encap:Ethernet HWaddr 02:42:ac:11:00:01
```

```
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:1/64 Scope:Link
          inet6 addr: 1553:ba3:2::242:ac11:1/64 Scope:Global
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:10 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:738 (738.0 B)  TX bytes:836 (836.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

All the traffic to the `1553:ba3:2::/64` subnet will be routed via the `docker0` interface.

The preceding container is assigned using `fe80::42:acff:fe11:1/64` as the link-local address and `1553:ba3:2::242:ac11:1/64` as the global routable IPv6 address.

Note

Link-local and loopback addresses have link-local scope, which means they are to be used in a directly attached network (link). All other addresses have global (or universal) scope, which means they are globally routable and can be used to connect to addresses with global scope anywhere.

Configuring a DNS server

Docker provides hostname and DNS configurations for each container without us having to build a custom image. It overlays the `/etc` folder inside the container with virtual files, in which it can write new information.

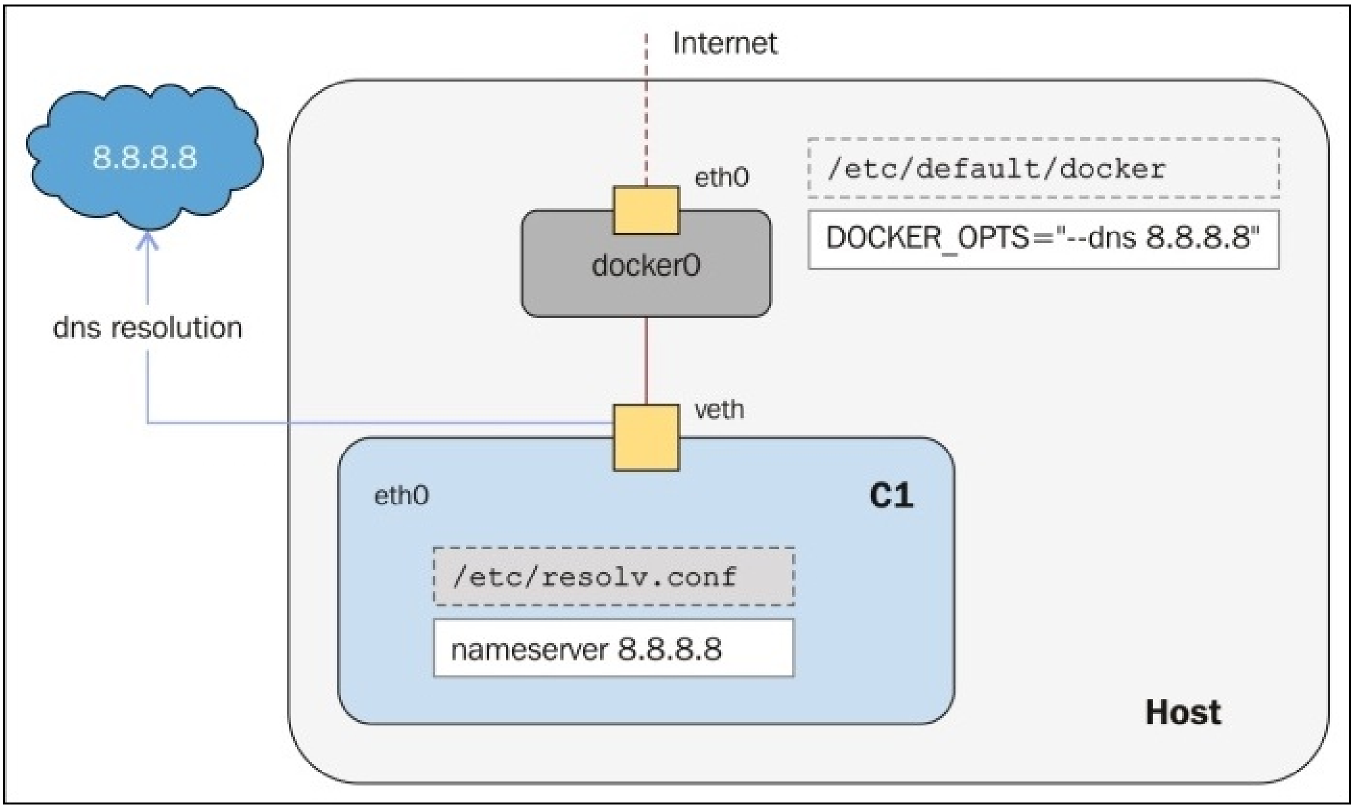
This can be seen by running the `mount` command inside the container. Containers receive the same `resolv.conf` file as that of the host machine when they are created initially. If a host's `resolv.conf` file is modified, this will be reflected in the container's `/resolv.conf` file only when the container is restarted.

In Docker, you can set DNS options in two ways:

- Using `docker run --dns=<ip-address>`
- Adding `DOCKER_OPTS="--dns ip-address"` to the Docker daemon file

You can also specify the search domain using `--dns-search=<DOMAIN>`.

The following figure shows a **nameserver** being configured in a container using the `DOCKER_OPTS` setting in the Docker daemon file:


Internet
8.8.8.8
eth0
/etc/default/docker
DOCKER_OPTS="--dns 8.8.8.8"
docker0
dns resolution
veth
eth0
C1
/etc/resolv.conf
nameserver 8.8.8.8
Host

The main DNS files are as follows:

- `/etc/hostname`
- `/etc/resolv.conf`
- `/etc/hosts`

The following is the command to add a DNS server:

```
# docker run --dns=8.8.8.8 --net="bridge" -t -i  ubuntu:latest /bin/bash
```

Add hostnames using the following command:

```
#docker run --dns=8.8.8.8 --hostname=docker-vm1  -t -i  ubuntu:latest
/bin/bash
```

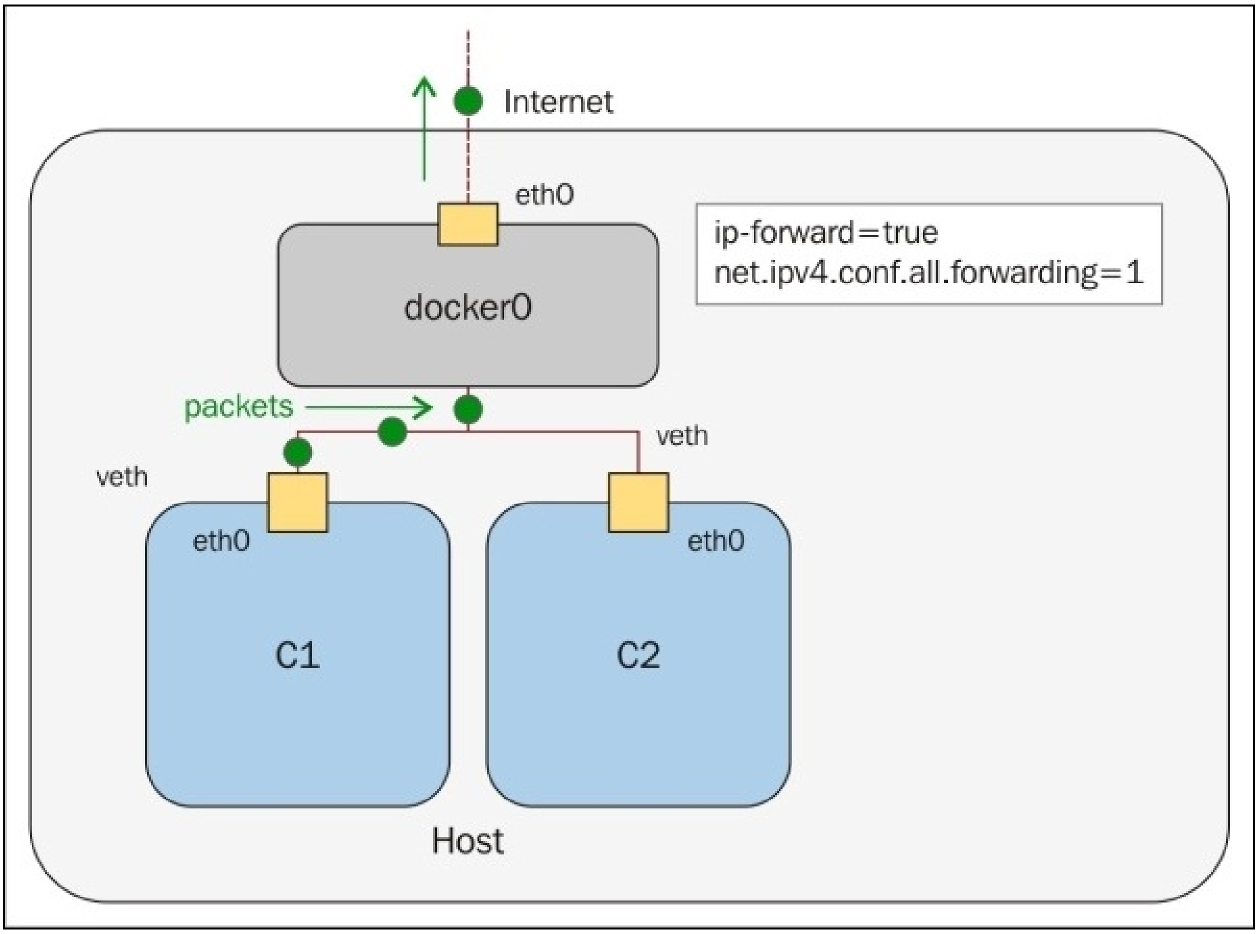
Communication between containers and external networks

Packets can only pass between containers if the `ip_forward` parameter is set to `1`. Usually, you will simply leave the Docker server at its default setting, `--ip-forward=true`, and Docker will set `ip_forward` to `1` for you when the server starts up.

To check the settings or to turn IP forwarding on manually, use these commands:

```
# cat /proc/sys/net/ipv4/ip_forward
0
# echo 1 > /proc/sys/net/ipv4/ip_forward
# cat /proc/sys/net/ipv4/ip_forward
1
```

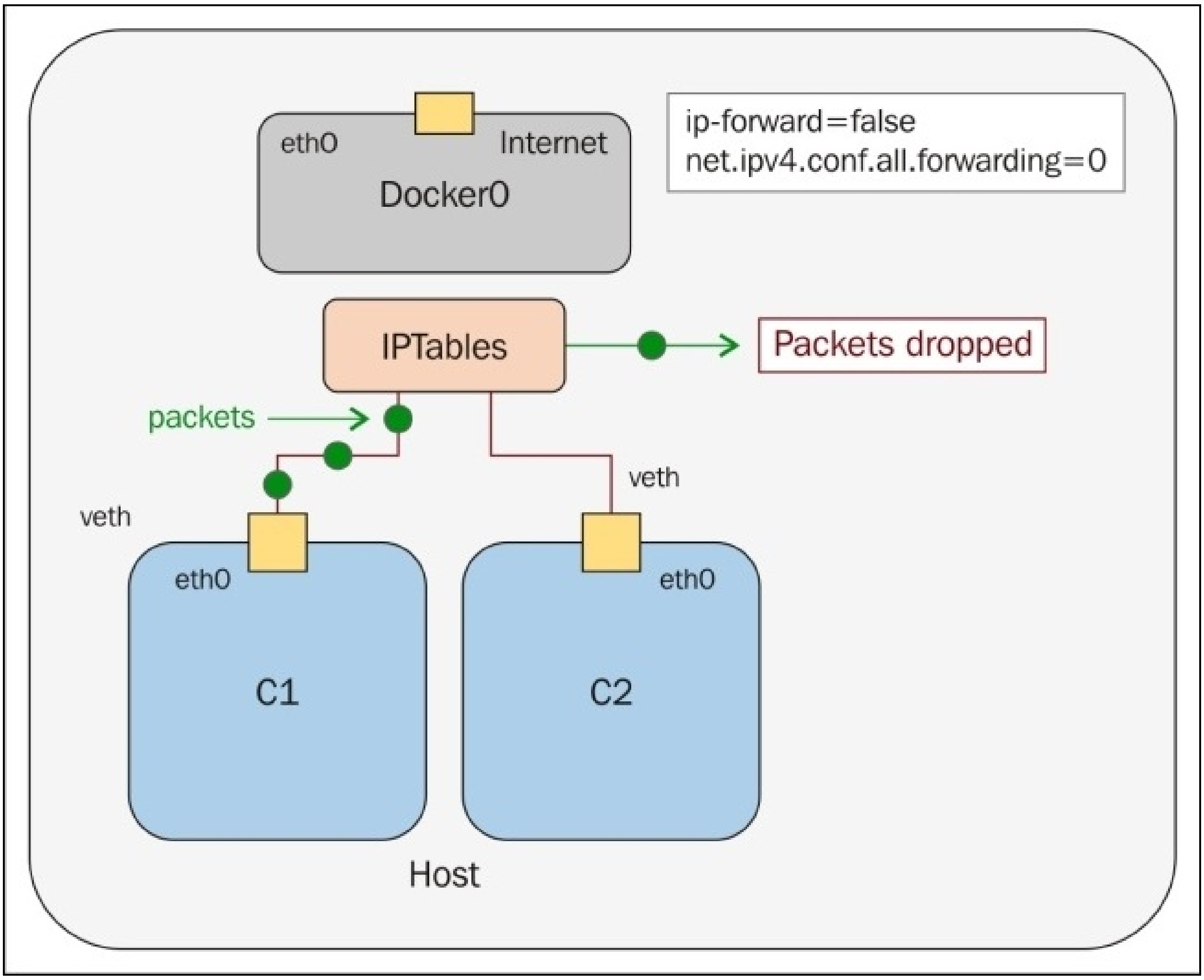
By enabling `ip_forward`, users can make communication possible between containers and the external world; it will also be required for inter-container communication if you are in a multiple-bridge setup. The following figure shows how `ip_forward = false` forwards all the packets to/from the container from/to the external network:



Docker will not delete or modify any pre-existing rules from the Docker filter chain. This allows users to create rules to restrict access to containers.

Docker uses the `docker0` bridge for packet flow between all the containers on a single host. It adds a rule to forward the chain using IPTables in order for the packets to flow between two containers. Setting `--icc=false` will drop all the packets.

When the Docker daemon is configured with both `--icc=false` and `--iptables=true` and `docker run` is invoked with the `--link` option, the Docker server will insert a pair of IPTables accept rules for new containers to connect to the ports exposed by the other containers, which will be the ports that have been mentioned in the exposed lines of its Dockerfile. The following figure shows how `ip_forward = false` drops all the packets to/from the container from/to the external network:


ip-forward=false
net.ipv4.conf.all.forwarding=0
eth0
Internet
Docker0
IPTables
Packets dropped
packets
veth
veth
eth0
eth0
C1
C2
Host

By default, Docker's `forward` rule permits all external IPs. To allow only a specific IP or network to access the containers, insert a negated rule at the top of the Docker filter chain.

For example, using the following command, you can restrict external access such that only the source IP `10.10.10.10` can access the containers:

```
#iptables –I DOCKER –i ext_if ! –s 10.10.10.10 –j DROP
```

Restricting SSH access from one container to another

Following these steps to restrict SSH access from one container to another:

1. Create two containers, `c1` and `c2`.

   For `c1`, use the following command:

```
# docker run -i -t --name c1 ubuntu:latest /bin/bash
```

The output generated is as follows:

```
root@7bc2b6cb1025:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:05
          inet addr:172.17.0.5  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: 2001:db8:1::242:ac11:5/64 Scope:Global
          inet6 addr: fe80::42:acff:fe11:5/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:738 (738.0 B)  TX bytes:696 (696.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

For `c2`, use the following command:

```
# docker run -i -t --name c2 ubuntu:latest /bin/bash
```

The following is the output generated:

```
root@e58a9bf7120b:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:06
          inet addr:172.17.0.6  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: 2001:db8:1::242:ac11:6/64 Scope:Global
          inet6 addr: fe80::42:acff:fe11:6/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:6 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:696 (696.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

We can test connectivity between the containers using the IP address we’ve just discovered. Let’s see this now using the `ping` tool:

```
root@7bc2b6cb1025:/# ping 172.17.0.6
PING 172.17.0.6 (172.17.0.6) 56(84) bytes of data.
64 bytes from 172.17.0.6: icmp_seq=1 ttl=64 time=0.139 ms
64 bytes from 172.17.0.6: icmp_seq=2 ttl=64 time=0.110 ms
^C
--- 172.17.0.6 ping statistics ---
```

```
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.110/0.124/0.139/0.018 ms
root@7bc2b6cb1025:/#

root@e58a9bf7120b:/# ping 172.17.0.5
PING 172.17.0.5 (172.17.0.5) 56(84) bytes of data.
64 bytes from 172.17.0.5: icmp_seq=1 ttl=64 time=0.270 ms
64 bytes from 172.17.0.5: icmp_seq=2 ttl=64 time=0.107 ms
^C
--- 172.17.0.5 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 1002ms
rtt min/avg/max/mdev = 0.107/0.188/0.270/0.082 ms
root@e58a9bf7120b:/#
```

2. Install `openssh-server` on both the containers:

   ```
   #apt-get install openssh-server
   ```

3. Enable iptables on the host machine:
   1. Initially, you will be able to SSH from one container to another.
   2. Stop the Docker service and add `DOCKER_OPTS="--icc=false --iptables=true"` to the default Dockerfile of the host machine. This option will enable the iptables firewall and drop all ports between the containers.

      By default, `iptables` is not enabled on the host. Use the following command to enable it:

      ```
      root@ubuntu:~# iptables -L -n
      Chain INPUT (policy ACCEPT)
      target     prot opt source               destination
      Chain FORWARD (policy ACCEPT)
      target     prot opt source               destination
      DOCKER     all — 0.0.0.0/0            0.0.0.0/0
      ACCEPT     all — 0.0.0.0/0            0.0.0.0/0            ctstate
      RELATED,ESTABLISHED
      ACCEPT     all — 0.0.0.0/0            0.0.0.0/0
      DOCKER     all — 0.0.0.0/0            0.0.0.0/0
      ACCEPT     all — 0.0.0.0/0            0.0.0.0/0            ctstate
      RELATED,ESTABLISHED
      ACCEPT     all — 0.0.0.0/0            0.0.0.0/0
      ACCEPT     all — 0.0.0.0/0            0.0.0.0/0
      ACCEPT     all — 0.0.0.0/0            0.0.0.0/0

      #service docker stop
      #vi /etc/default/docker
      ```

   3. Docker Upstart and SysVinit configuration file. Customize the location of the Docker binary (especially for development testing):

      ```
      #DOCKER="/usr/local/bin/docker"
      ```

   4. Use `DOCKER_OPTS` to modify the daemon’s startup options:

      ```
      #DOCKER_OPTS="--dns 8.8.8.8 --dns 8.8.4.4"
      #DOCKER_OPTS="--icc=false --iptables=true"
      ```

    5. Restart the Docker service:

       ```
       # service docker start
       ```

    6. Inspect `iptables`:

       ```
       root@ubuntu:~# iptables -L -n
       Chain INPUT (policy ACCEPT)
       target     prot opt source             destination
       Chain FORWARD (policy ACCEPT)
       target     prot opt source             destination
       DOCKER     all — 0.0.0.0/0          0.0.0.0/0
       ACCEPT     all — 0.0.0.0/0          0.0.0.0/0    ctstate RELATED,
       ESTABLISHED
       ACCEPT     all — 0.0.0.0/0          0.0.0.0/0
       DOCKER     all — 0.0.0.0/0          0.0.0.0/0
       ACCEPT     all — 0.0.0.0/0          0.0.0.0/0   ctstate RELATED,
       ESTABLISHED
       ACCEPT     all — 0.0.0.0/0          0.0.0.0/0
       ACCEPT     all — 0.0.0.0/0          0.0.0.0/0
       DROP       all — 0.0.0.0/0          0.0.0.0/0
       ```

   The `DROP` rule has been added to iptables on the host machine, which drops a connection between containers. Now you will be unable to SSH between the containers.

4. We can communicate with or connect containers using the `--link` parameter, with the help of following steps:

    1. Create the first container, which will act as the server, `sshserver`:

       ```
       root@ubuntu:~# docker run -i -t -p 2222:22 --name sshserver ubuntu
       bash
       root@9770be5acbab:/#
       ```

    2. Execute the `iptables` command, and you will find a Docker chain rule added:

       ```
       #root@ubuntu:~# iptables -L -n
       Chain INPUT (policy ACCEPT)
       target     prot opt source         destination
       Chain FORWARD (policy ACCEPT)
       target     prot opt source         destination
       Chain OUTPUT (policy ACCEPT)
       target     prot opt source         destination
       Chain DOCKER (0 references)
       target     prot opt source         destination
       ACCEPT     tcp — 0.0.0.0/0        172.17.0.3     tcp dpt:22
       ```

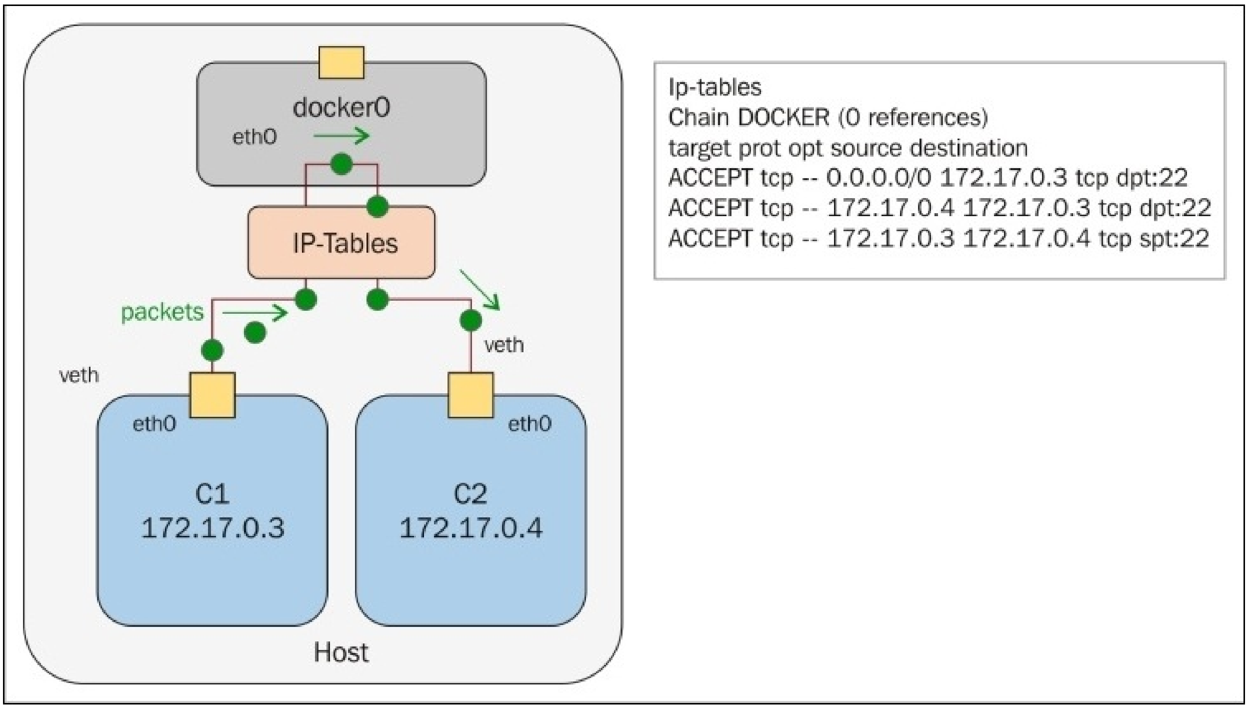
    3. Create the second container, which acts like a client, `sshclient`:

       ```
       root@ubuntu:~# docker run -i -t --name sshclient --link
       sshserver:sshserver ubuntu bash
       root@979d46c5c6a5:/#
       ```

    4. We can see that there are more rules added to the Docker chain rule:

```
root@ubuntu:~# iptables -L -n
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
Chain DOCKER (0 references)
target     prot opt source               destination
ACCEPT     tcp — 0.0.0.0/0            172.17.0.3           tcp
dpt:22
ACCEPT     tcp — 172.17.0.4           172.17.0.3           tcp
dpt:22
ACCEPT     tcp — 172.17.0.3           172.17.0.4           tcp
spt:22
root@ubuntu:~#
```

The following image explains communication between the containers using the `--link` flag:



```
Ip-tables
Chain DOCKER (0 references)
target prot opt source destination
ACCEPT tcp -- 0.0.0.0/0 172.17.0.3 tcp dpt:22
ACCEPT tcp -- 172.17.0.4 172.17.0.3 tcp dpt:22
ACCEPT tcp -- 172.17.0.3 172.17.0.4 tcp spt:22
```

5. You can inspect your linked container with the `docker inspect` command:

```
root@ubuntu:~# docker inspect -f "{{ .HostConfig.Links }}"
sshclient
[/sshserver:/sshclient/sshserver]
```

Now you can successfully ssh into sshserver with its IP.

```
#ssh root@172.17.0.3 –p 22
```

Using the `--link` parameter, Docker creates a secure channel between the containers that doesn’t need to expose any ports externally on the containers.

Configuring the Docker bridge

The Docker server creates a bridge called `docker0` by default inside the Linux kernel, and it can pass packets back and forth between other physical or virtual network interfaces so that they behave as a single Ethernet network . Run the following command to find out the list of interfaces in a VM and the IP addresses they are connected to:

```
root@ubuntu:~# ifconfig
docker0   Link encap:Ethernet  HWaddr 56:84:7a:fe:97:99
          inet addr:172.17.42.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::5484:7aff:fefe:9799/64 Scope:Link
          inet6 addr: fe80::1/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:11909 errors:0 dropped:0 overruns:0 frame:0
          TX packets:14826 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:516868 (516.8 KB)  TX bytes:46460483 (46.4 MB)
eth0      Link encap:Ethernet  HWaddr 00:0c:29:0d:f4:2c
          inet addr:192.168.186.129  Bcast:192.168.186.255
Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe0d:f42c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:108865 errors:0 dropped:0 overruns:0 frame:0
          TX packets:31708 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:59902195 (59.9 MB)  TX bytes:3916180 (3.9 MB)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:336 (336.0 B)  TX bytes:336 (336.0 B)
```

Once you have one or more containers up and running, you can confirm that Docker has properly connected them to the `docker0` bridge by running the `brctl` command on the host machine and looking at the `interfaces` column of the output.

Before configuring the `docker0` bridge, install the bridge utilities:

```
# apt-get install bridge-utils
```

Here is a host with two different containers connected:

```
root@ubuntu:~# brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.56847afe9799       no              veth21b2e16
                                                        veth7092a45
```

Docker uses the `docker0` bridge settings whenever a container is created. It assigns a new IP address from the range available on the bridge whenever a new container is created, as can be seen here:

```
root@ubuntu:~# docker run -t -i --name container1 ubuntu:latest /bin/bash
root@e54e9312dc04:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:07
          inet addr:172.17.0.7  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: 2001:db8:1::242:ac11:7/64 Scope:Global
          inet6 addr: fe80::42:acff:fe11:7/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:738 (738.0 B)  TX bytes:696 (696.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
root@e54e9312dc04:/# ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0  proto kernel  scope link  src 172.17.0.7
```

By default, Docker provides a virtual network called `docker0`, which has the IP address `172.17.42.1`. Docker containers have IP addresses in the range of `172.17.0.0/16`.

To change the default settings in Docker, modify the file `/etc/default/docker`.

Changing the default bridge from `docker0` to `br0` can be done like this:

```
# sudo service docker stop
# sudo ip link set dev docker0 down
# sudo brctl delbr docker0
# sudo iptables -t nat -F POSTROUTING
# echo 'DOCKER_OPTS="-b=br0"' >> /etc/default/docker
# sudo brctl addbr br0
# sudo ip addr add 192.168.10.1/24 dev br0
# sudo ip link set dev br0 up
# sudo service docker start
```

The following command displays the new bridge name and the IP address range of the Docker service:

```
root@ubuntu:~# ifconfig
br0       Link encap:Ethernet  HWaddr ae:b2:dc:ed:e6:af
          inet addr:192.168.10.1  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::acb2:dcff:feed:e6af/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:738 (738.0 B)
eth0      Link encap:Ethernet  HWaddr 00:0c:29:0d:f4:2c
          inet addr:192.168.186.129  Bcast:192.168.186.255
Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe0d:f42c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:110823 errors:0 dropped:0 overruns:0 frame:0
          TX packets:33148 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:60081009 (60.0 MB)  TX bytes:4176982 (4.1 MB)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:336 (336.0 B)  TX bytes:336 (336.0 B)
```

Overlay networks and underlay networks

An overlay is a virtual network that is built on top of underlying network infrastructure (the underlay). The purpose is to implement a network service that is not available in the physical network.

Network overlay dramatically increases the number of virtual subnets that can be created on top of the physical network, which in turn supports multi-tenancy and virtualization.

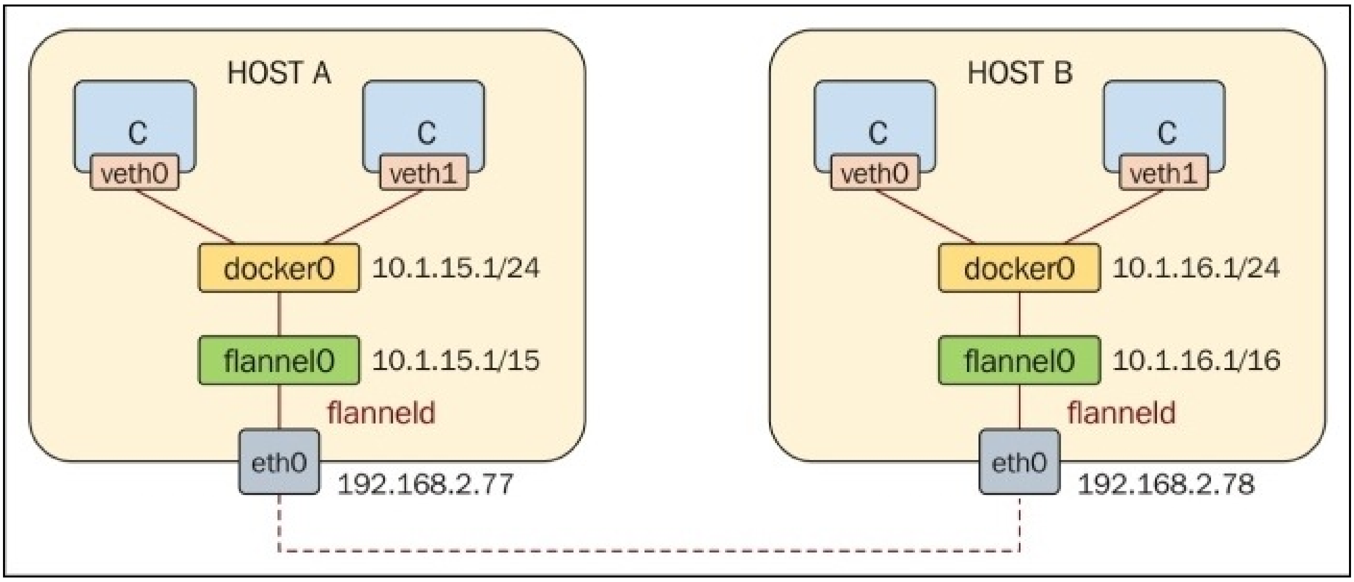
Every container in Docker is assigned an IP address, which is used for communication with other containers. If a container has to communicate with the external network, you set up networking in the host system and expose or map the port from the container to the host machine. With this, applications running inside containers will not be able to advertise their external IP and ports, as the information will not be available to them.

The solution is to somehow assign unique IPs to each Docker container across all hosts and have some networking product that routes traffic between hosts.

There are different projects to deal with Docker networking, as follows:

- Flannel
- Weave
- Open vSwitch

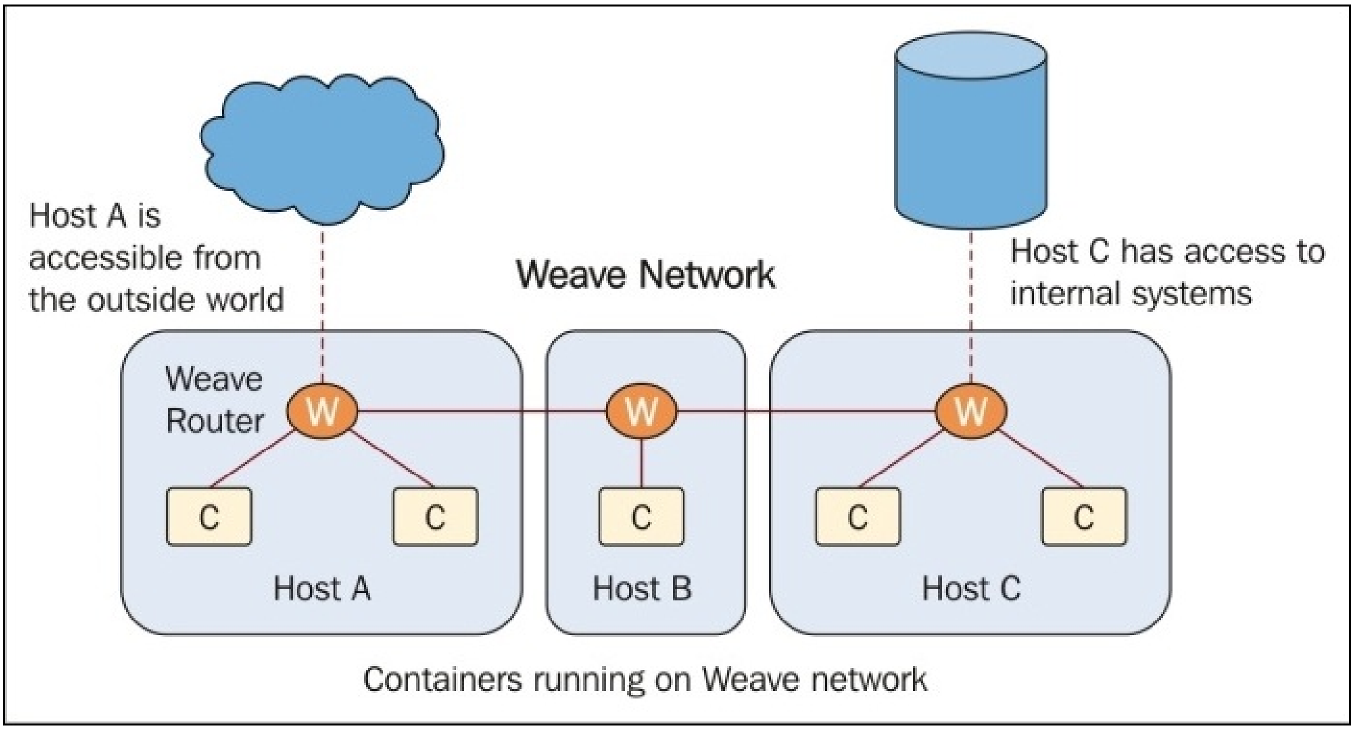
Flannel provides a solution by giving each container an IP that can be used for container-to-container communication. Using packet encapsulation, it creates a virtual overlay network over the host network. By default, Flannel provides a `/24` subnet to hosts, from which the Docker daemon allocates IPs to containers. The following figure shows the communication between containers using Flannel:



Flannel runs an agent, **flanneld**, on each host and is responsible for allocating a subnet lease out of a preconfigured address space. Flannel uses etcd to store the network configuration, allocated subnets, and auxiliary data (such as the host's IP).

Flannel uses the universal TUN/TAP device and creates an overlay network using UDP to encapsulate IP packets. Subnet allocation is done with the help of etcd, which maintains the overlay subnet-to-host mappings.

Weave creates a virtual network that connects Docker containers deployed across hosts/VMs and enables their automatic discovery. The following figure shows a Weave network:


Host A is accessible from the outside world
Weave Network
Host C has access to internal systems
Weave Router
W
W
W
C
C
C
C
C
Host A
Host B
Host C
Containers running on Weave network

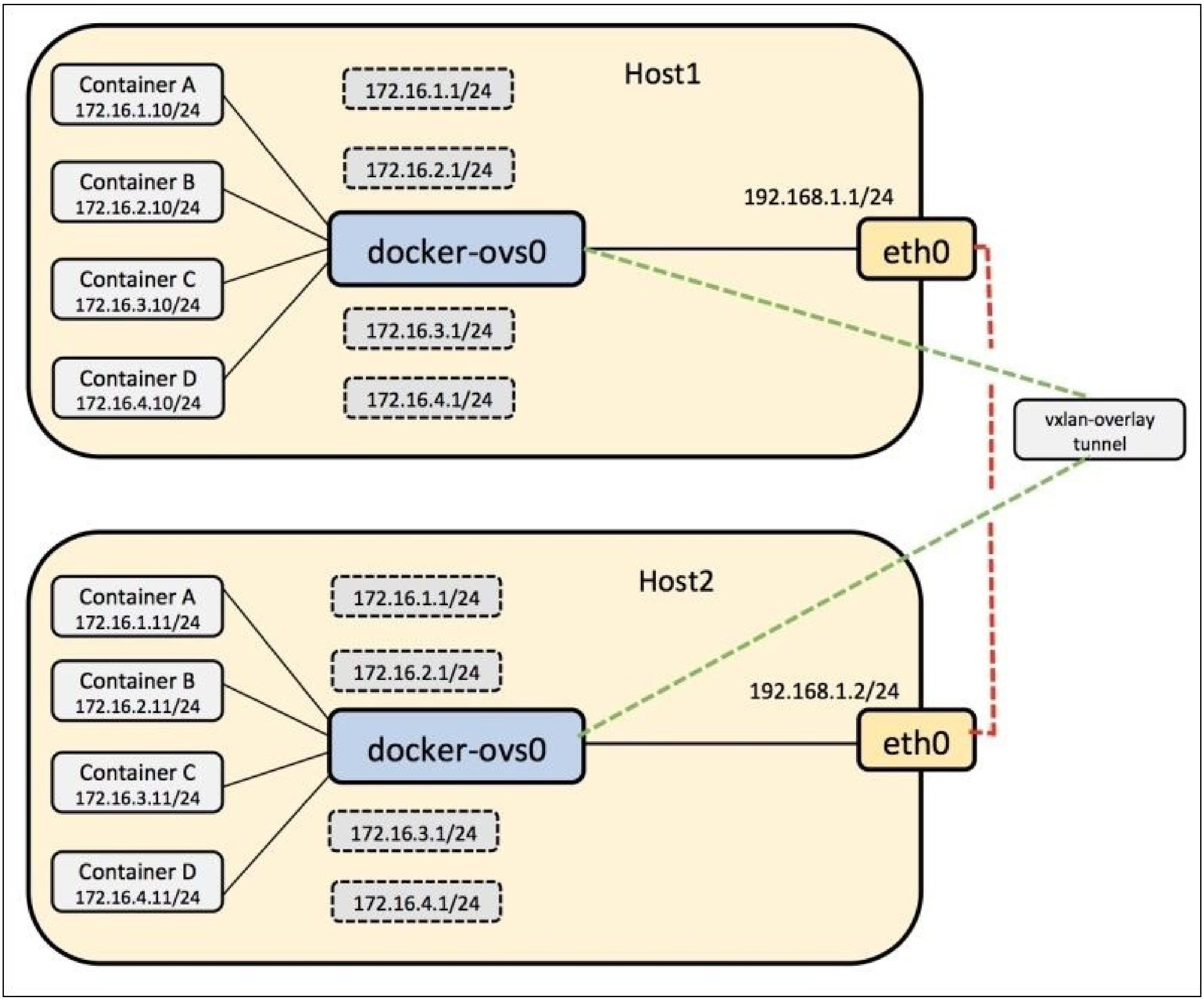
Weave can traverse firewalls and operate in partially connected networks. Traffic can be optionally encrypted, allowing hosts/VMs to be connected across an untrusted network.

Weave augments Docker's existing (single host) networking capabilities, such as the `docker0` bridge, so these can continue to be used by containers.

Open vSwitch is an open source OpenFlow-capable virtual switch that is typically used with hypervisors to interconnect virtual machines within a host and between different hosts across networks. Overlay networks need to create a virtual datapath using supported tunneling encapsulations, such as VXLAN and GRE.

The overlay datapath is provisioned between tunnel endpoints residing in the Docker host, which gives the appearance of all hosts within a given provider segment being directly connected to one another.

As a new container comes online, the prefix is updated in the routing protocol, announcing its location via a tunnel endpoint. As the other Docker hosts receive the updates, the forwarding rule is installed into the OVS for the tunnel endpoint that the host resides on. When the host is de-provisioned, a similar process occurs and tunnel endpoint Docker hosts remove the forwarding entry for the de-provisioned container. The following figure shows the communication between containers running on multiple hosts through OVS-based VXLAN tunnels:


Host1
Container A
172.16.1.10/24
Container B
172.16.2.10/24
Container C
172.16.3.10/24
Container D
172.16.4.10/24
172.16.1.1/24
172.16.2.1/24
172.16.3.1/24
172.16.4.1/24
docker-ovs0
192.168.1.1/24
eth0
vxlan-overlay tunnel
Host2
Container A
172.16.1.11/24
Container B
172.16.2.11/24
Container C
172.16.3.11/24
Container D
172.16.4.11/24
172.16.1.1/24
172.16.2.1/24
172.16.3.1/24
172.16.4.1/24
docker-ovs0
192.168.1.2/24
eth0

Summary

In this chapter, we discussed Docker's internal networking architecture. We learned about IPv4, IPv6, and DNS configuration in Docker. Later in the chapter, we covered the Docker bridge and communication between containers within a single host and in multiple hosts.

We also discussed overlay tunneling and different methods that are implemented in Docker networking, such as OVS, Flannel, and Weave.

In the next chapter, we will learn hands-on Docker networking, clubbed with various frameworks.

www.allitebooks.com

Chapter 3. Building Your First Docker Network

This chapter describes practical examples of Docker networking, spanning multiple containers over multiple hosts. We will cover the following topics:

- Introduction to Pipework
- Multiple containers over multiple hosts
- Towards scaling networks – introducing Open vSwitch
- Networking with overlay networks – Flannel
- Comparison of Docker networking options

Introduction to Pipework

Pipework lets you connect together containers in arbitrarily complex scenarios.

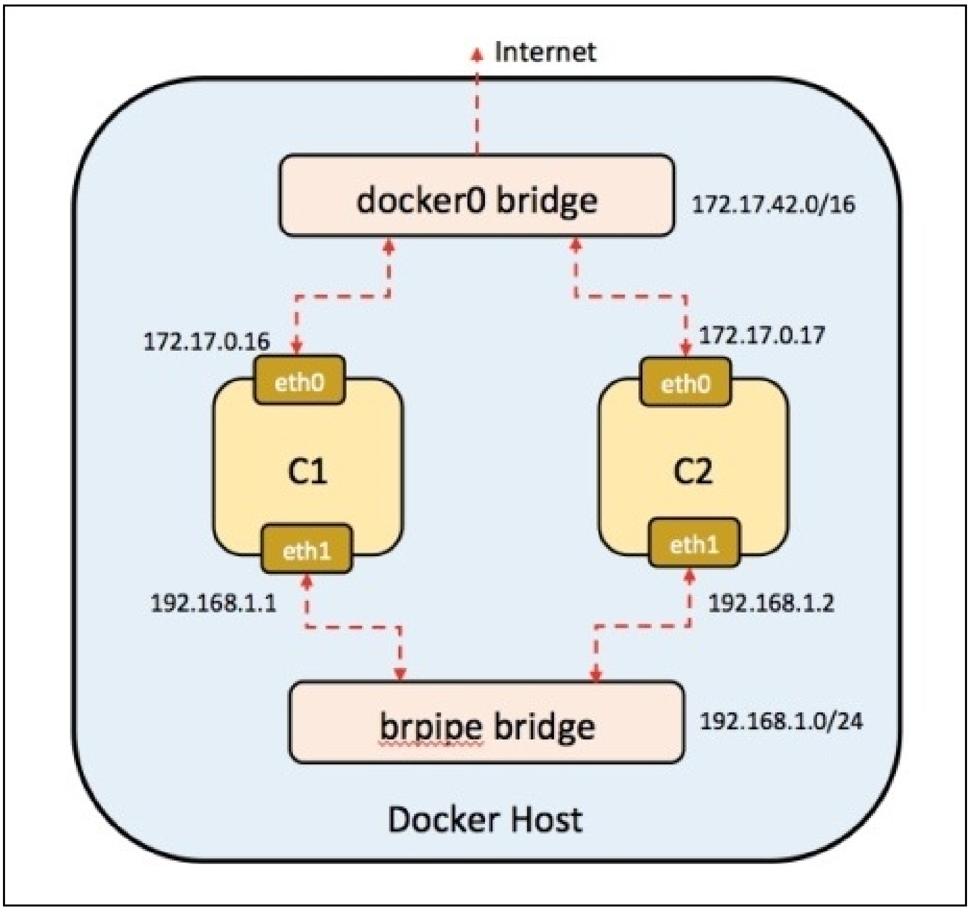
In practical terms, it creates a legacy Linux bridge, adds a new interface to the container, and then attaches the interface to that bridge; containers get a network segment on which to communicate with each other.

Multiple containers over a single host

Pipework is a shell script and installing it is simple:

```
#sudo wget -O /usr/local/bin/pipework
https://raw.githubusercontent.com/jpetazzo/pipework/master/pipework && sudo
chmod +x /usr/local/bin/pipework
```

The following figure shows container communication using Pipework:


Internet
docker0 bridge
172.17.42.0/16
172.17.0.16
172.17.0.17
eth0
eth0
C1
C2
eth1
eth1
192.168.1.1
192.168.1.2
brpipe bridge
192.168.1.0/24
Docker Host

First, create two containers:

```
#docker run -i -t --name c1 ubuntu:latest /bin/bash
root@5afb44195a69:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:10
          inet addr:172.17.0.16  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:10/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:13 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1038 (1.0 KB)  TX bytes:738 (738.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

#docker run -i -t --name c2 ubuntu:latest /bin/bash
root@c94d53a76a9b:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:11
          inet addr:172.17.0.17  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:11/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:738 (738.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Now let’s use Pipework to connect them:

```
#sudo pipework brpipe c1 192.168.1.1/24
```

This command creates a bridge, `brpipe`, on the host machine. It adds an `eth1` interface to the container `c1` with the IP address `192.168.1.1` and attaches the interface to the bridge as follows:

```
root@5afb44195a69:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:10
          inet addr:172.17.0.16  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:10/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:13 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1038 (1.0 KB)  TX bytes:738 (738.0 B)
eth1      Link encap:Ethernet  HWaddr ce:72:c5:12:4a:1a
          inet addr:192.168.1.1  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::cc72:c5ff:fe12:4a1a/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:23 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1806 (1.8 KB)  TX bytes:690 (690.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

```
#sudo pipework brpipe c2 192.168.1.2/24
```

This command will not create bridge `brpipe` as it already exists. It will add an `eth1` interface to the container `c2` and connect it to the bridge as follows:

```
root@c94d53a76a9b:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:11
          inet addr:172.17.0.17  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:11/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:648 (648.0 B)  TX bytes:738 (738.0 B)
eth1      Link encap:Ethernet  HWaddr 36:86:fb:9e:88:ba
          inet addr:192.168.1.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::3486:fbff:fe9e:88ba/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:8 errors:0 dropped:0 overruns:0 frame:0
          TX packets:9 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:648 (648.0 B)  TX bytes:690 (690.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Now the containers are connected and will be able to ping each other as they are on the same subnet, `192.168.1.0/24`. Pipework provides the advantage of adding static IP addresses to the containers.

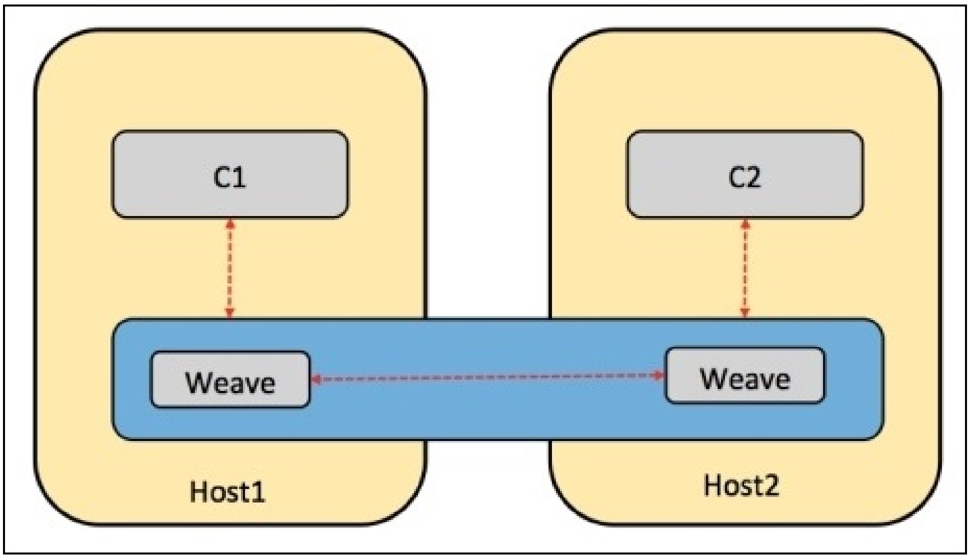
Weave your containers

Weave creates a virtual network that can connect Docker containers across multiple hosts as if they are all connected to a single switch. The Weave router itself runs as a Docker container and can encrypt routed traffic for transmission over the Internet. Services provided by application containers on the Weave network can be made accessible to the outside world, regardless of where those containers are running.

Use the following code to install Weave:

```
#sudo curl -L git.io/weave -o /usr/local/bin/weave
#sudo chmod a+x /usr/local/bin/weave
```

The following figure shows multihost communication using Weave:



On `$HOST1`, we run the following:

```
# weave launch
# eval $(weave proxy-env)
# docker run --name c1 -ti ubuntu
```

Next, we repeat similar steps on `$HOST2`:

```
# weave launch $HOST1
# eval $(weave proxy-env)
# docker run --name c2 -ti ubuntu
```

In the container started on `$HOST1`, the following output is generated:

```
root@c1:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:21
          inet addr:172.17.0.33  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:21/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:38 errors:0 dropped:0 overruns:0 frame:0
          TX packets:34 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3166 (3.1 KB)  TX bytes:2299 (2.2 KB)
ethwe     Link encap:Ethernet  HWaddr aa:99:8a:d5:4d:d4
          inet addr:10.128.0.3  Bcast:0.0.0.0  Mask:255.192.0.0
          inet6 addr: fe80::a899:8aff:fed5:4dd4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:65535  Metric:1
          RX packets:130 errors:0 dropped:0 overruns:0 frame:0
          TX packets:74 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:11028 (11.0 KB)  TX bytes:6108 (6.1 KB)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

You can see the Weave network interface, `ethwe`, using the `ifconfig` command:

```
root@c2:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:11:00:04
          inet addr:172.17.0.4  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:4/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:28 errors:0 dropped:0 overruns:0 frame:0
          TX packets:29 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:2412 (2.4 KB)  TX bytes:2016 (2.0 KB)
ethwe     Link encap:Ethernet  HWaddr 8e:7c:17:0d:0e:03
          inet addr:10.160.0.1  Bcast:0.0.0.0  Mask:255.192.0.0
          inet6 addr: fe80::8c7c:17ff:fe0d:e03/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:65535  Metric:1
          RX packets:139 errors:0 dropped:0 overruns:0 frame:0
          TX packets:74 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:11718 (11.7 KB)  TX bytes:6108 (6.1 KB)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

#root@c1:/# ping -c 1 -q c2
PING c2.weave.local (10.160.0.1) 56(84) bytes of data.
--- c2.weave.local ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.317/1.317/1.317/0.000 ms
```

Similarly, in the container started on `$HOST2`, the following output is generated:

```
#root@c2:/# ping -c 1 -q c1
PING c1.weave.local (10.128.0.3) 56(84) bytes of data.
--- c1.weave.local ping statistics ---
```

```
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 1.658/1.658/1.658/0.000 ms
```

So there we have it—two containers on separate hosts happily talking to each other.

Open vSwitch

Docker uses the Linux bridge `docker0` by default. However, there are cases where **Open vSwitch** (**OVS**) might be required instead of a Linux bridge. A single Linux bridge can only handle 1024 ports – this limits the scalability of Docker as we can only create 1024 containers, each with a single network interface.

Single host OVS

We will now install OVS on a single host, create two containers, and connect them to an OVS bridge.

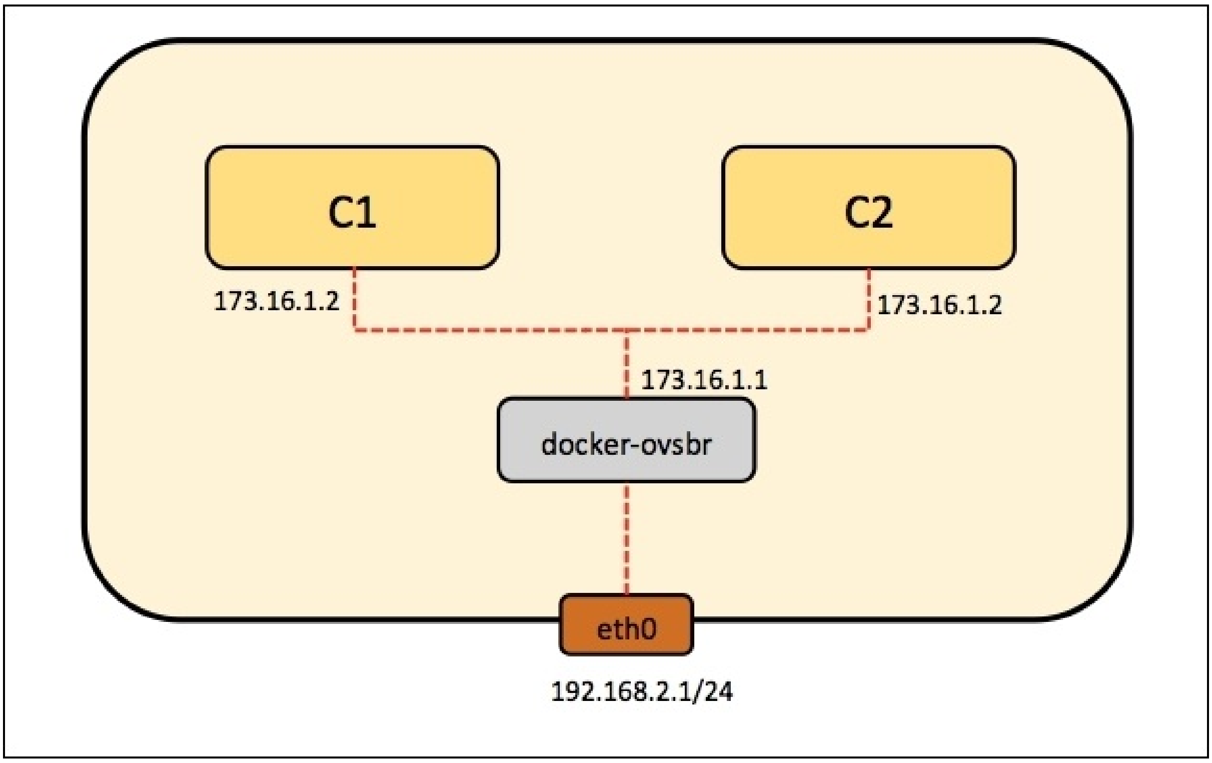
Use this command to install OVS:

```
# sudo apt-get install openvswitch-switch
```

Install the `ovs-docker` utility with the following:

```
# cd /usr/bin
# wget
https://raw.githubusercontent.com/openvswitch/ovs/master/utilities/ovs-
docker
# chmod a+rwx ovs-docker
```

The following diagram shows the single-host OVS:



Creating an OVS bridge

Here, we will be adding a new OVS bridge and configuring it so that we can get the containers connected on a different network, as follows:

```
# ovs-vsctl add-br ovs-br1
# ifconfig ovs-br1 173.16.1.1 netmask 255.255.255.0 up
```

Add a port from the OVS bridge to the Docker container using the following steps:

1. Create two Ubuntu Docker containers:

   ```
   # docker run -I -t --name container1 ubuntu /bin/bash
   # docekr run -I -t --name container2 ubuntu /bin/bash
   ```

2. Connect the container to the OVS bridge:

```
# ovs-docker add-port ovs-br1 eth1 container1 --ipaddress=173.16.1.2/24
# ovs-docker add-port ovs-br1 eth1 container2 --ipaddress=173.16.1.3/24
```

3. Test the connection between the two containers connected via an OVS bridge using the `ping` command. First, find out their IP addresses:

```
# docker exec container1 ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:10:11:02
          inet addr:172.16.17.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:acff:fe10:1102/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1472  Metric:1
          RX packets:36 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4956 (4.9 KB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)

# docker exec container2 ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:ac:10:11:03
          inet addr:172.16.17.3  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:acff:fe10:1103/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1472  Metric:1
          RX packets:27 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4201 (4.2 KB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Now that we know the IP addresses of `container1` and `container2`, we can ping them:

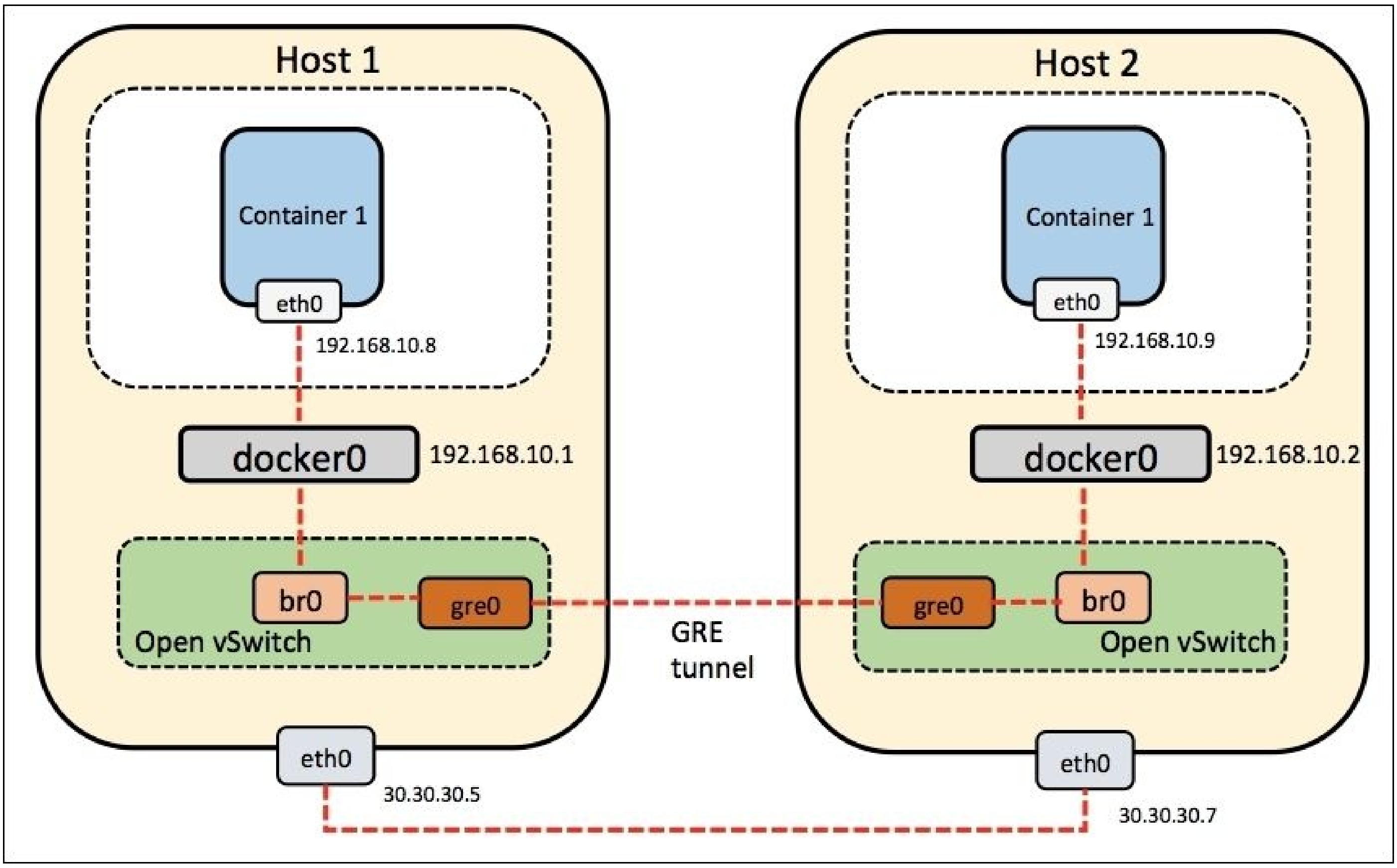
```
# docker exec container2 ping 172.16.17.2
PING 172.16.17.2 (172.16.17.2) 56(84) bytes of data.
64 bytes from 172.16.17.2: icmp_seq=1 ttl=64 time=0.257 ms
64 bytes from 172.16.17.2: icmp_seq=2 ttl=64 time=0.048 ms
64 bytes from 172.16.17.2: icmp_seq=3 ttl=64 time=0.052 ms

# docker exec container1 ping 172.16.17.2
```

```
PING 172.16.17.2 (172.16.17.2) 56(84) bytes of data.
64 bytes from 172.16.17.2: icmp_seq=1 ttl=64 time=0.060 ms
64 bytes from 172.16.17.2: icmp_seq=2 ttl=64 time=0.035 ms
64 bytes from 172.16.17.2: icmp_seq=3 ttl=64 time=0.031 ms
```

Multiple host OVS

Let’s see how to connect Docker containers on multiple hosts using OVS.

Let’s consider our setup as shown in the following diagram, which contains two hosts, **Host 1** and **Host 2**, running Ubuntu 14.04:


Host 1
Container 1
eth0
192.168.10.8
docker0
192.168.10.1
br0
gre0
Open vSwitch
eth0
30.30.30.5
GRE
tunnel
Host 2
Container 1
eth0
192.168.10.9
docker0
192.168.10.2
gre0
br0
Open vSwitch
eth0
30.30.30.7

Install Docker and Open vSwitch on both the hosts:

```
# wget -qO- https://get.docker.com/ | sh
# sudo apt-get install openvswitch-switch
```

Install the `ovs-docker` utility:

```
# cd /usr/bin
# wget
https://raw.githubusercontent.com/openvswitch/ovs/master/utilities/ovs-
docker
# chmod a+rwx ovs-docker
```

By default, Docker chooses a random network to run its containers in. It creates a bridge, `docker0`, and assigns an IP address (`172.17.42.1`) to it. So, both **Host 1** and **Host 2** `docker0` bridge IP addresses are the same, due to which it is difficult for containers in both the hosts to communicate. To overcome this, let’s assign static IP addresses to the network, that is, `192.168.10.0/24`.

Let’s see how to change the default Docker subnet.

Execute the following commands on Host 1:

```
# service docker stop
# ip link set dev docker0 down
# ip addr del 172.17.42.1/16 dev docker0
# ip addr add 192.168.10.1/24 dev docker0
# ip link set dev docker0 up
# ip addr show docker0
# service docker start
```

Add the `br0` OVS bridge:

```
# ovs-vsctl add-br br0
```

Create the tunnel to the other host and attach it to the:

```
# add-port br0 gre0—set interface gre0 type=gre
options:remote_ip=30.30.30.8
```

Add the `br0` bridge to the `docker0` bridge:

```
# brctl addif docker0 br0
```

Execute the following commands on Host 2:

```
# service docker stop
# iptables -t nat -F POSTROUTING
# ip link set dev docker0 down
# ip addr del 172.17.42.1/16 dev docker0
# ip addr add 192.168.10.2/24 dev docker0
# ip link set dev docker0 up
# ip addr show docker0
# service docker start
```

Add the `br0` OVS bridge:

```
# ip link set br0 up
# ovs-vsctl add-br br0
```

Create the tunnel to the other host and attach it to the:

```
# br0 bridge ovs-vsctl add-port br0 gre0—set interface gre0 type=gre
options:remote_ip=30.30.30.7
```

Add the `br0` bridge to the `docker0` bridge:

```
# brctl addif docker0 br0
```

The `docker0` bridge is attached to another bridge, `br0`. This time, it's an OVS bridge. This means that all traffic between the containers is routed through `br0` too.

Additionally, we need to connect together the networks from both the hosts in which the containers are running. A GRE tunnel is used for this purpose. This tunnel is attached to the `br0` OVS bridge and, as a result, to `docker0` too.

After executing the preceding commands on both hosts, you should be able to ping the `docker0` bridge addresses from both hosts.

On Host 1, the following output is generated on using the `ping` command:

```
# ping 192.168.10.2
PING 192.168.10.2 (192.168.10.2) 56(84) bytes of data.
64 bytes from 192.168.10.2: icmp_seq=1 ttl=64 time=0.088 ms
64 bytes from 192.168.10.2: icmp_seq=2 ttl=64 time=0.032 ms
^C
--- 192.168.10.2 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.032/0.060/0.088/0.028 ms
```

On Host 2, the following output is generated on using the `ping` command:

```
# ping 192.168.10.1
PING 192.168.10.1 (192.168.10.1) 56(84) bytes of data.
64 bytes from 192.168.10.1: icmp_seq=1 ttl=64 time=0.088 ms
64 bytes from 192.168.10.1: icmp_seq=2 ttl=64 time=0.032 ms
^C
--- 192.168.10.1 ping statistics ---
2 packets transmitted, 2 received, 0% packet loss, time 999ms
rtt min/avg/max/mdev = 0.032/0.060/0.088/0.028 ms
```

Let’s see how to create containers on both the hosts.

On Host 1, use the following code:

```
# docker run -t -i --name container1 ubuntu:latest /bin/bash
```

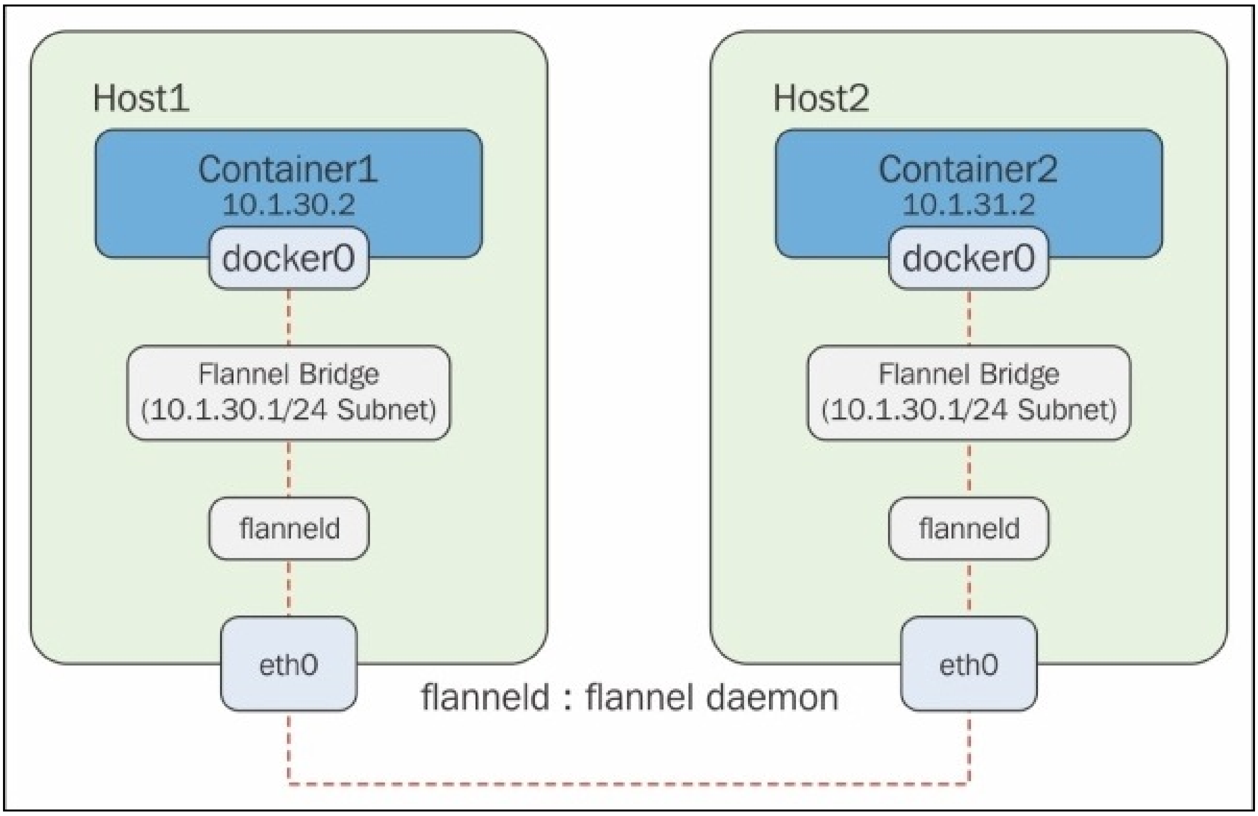
On Host 2, use the following code:

```
# docker run -t -i --name container2 ubuntu:latest /bin/bash
```

Now we can ping `container2` from `container1`. In this way, we connect Docker containers on multiple hosts using Open vSwitch.

Networking with overlay networks – Flannel

Flannel is the virtual network layer that provides the subnet to each host for use with Docker containers. It is packaged with CoreOS but can be configured on other Linux OSes as well. Flannel creates the overlay by actually connecting itself to Docker bridge, to which containers are attached, as shown in the following figure. To setup Flannel, two host machines or VMs are required, which can be CoreOS or, more preferably, Linux OS, as shown in this figure:


Host1
Container1
10.1.30.2
docker0
Flannel Bridge
(10.1.30.1/24 Subnet)
flanneld
eth0
Host2
Container2
10.1.31.2
docker0
Flannel Bridge
(10.1.30.1/24 Subnet)
flanneld
eth0
flanneld : flannel daemon

The Flannel code can be cloned from GitHub and built locally, if required, on a different flavor of Linux OS, as shown here. It comes preinstalled in CoreOS:

```
# git clone https://github.com/coreos/flannel.git
Cloning into 'flannel'...
remote: Counting objects: 2141, done.
remote: Compressing objects: 100% (19/19), done.
remote: Total 2141 (delta 6), reused 0 (delta 0), pack-reused 2122
Receiving objects: 100% (2141/2141), 4.
Checking connectivity… done.

# sudo docker run -v `pwd`:/opt/flannel -i -t google/golang /bin/bash -c
"cd /opt/flannel && ./build"
Building flanneld…
```

CoreOS machines can be easily configured using Vagrant and VirtualBox, as per the tutorial mentioned in the following link:

https://coreos.com/os/docs/latest/booting-on-vagrant.html

After the machines are created and logged in to, we will find a Flannel bridge automatically created using the `etcd` configuration:

```
# ifconfig flannel0
flannel0: flags=4305<UP,POINTOPOINT,RUNNING,NOARP,MULTICAST>  mtu 1472
          inet 10.1.30.0  netmask 255.255.0.0  destination 10.1.30.0
          unspec 00-00-00-00-00-00-00-00-00-00-00-00-00-00-00-00
txqueuelen 500 (UNSPEC)
          RX packets 243  bytes 20692 (20.2 KiB)
          RX errors 0  dropped 0  overruns 0  frame 0
          TX packets 304  bytes 25536 (24.9 KiB)
          TX errors 0  dropped 0 overruns 0  carrier 0  collisions 0
```

The Flannel environment can be checked by viewing `subnet.env`:

```
# cat /run/flannel/subnet.env
FLANNEL_NETWORK=10.1.0.0/16
FLANNEL_SUBNET=10.1.30.1/24
FLANNEL_MTU=1472
FLANNEL_IPMASQ=true
```

The Docker daemon requires to be restarted with the following commands in order to get the networking re-instantiated with the subnet from the Flannel bridge:

```
# source /run/flannel/subnet.env
# sudo rm /var/run/docker.pid
# sudo ifconfig docker0 ${FLANNEL_SUBNET}
# sudo docker -d --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU} & INFO[0000]
[graphdriver] using prior storage driver "overlay"
INFO[0000] Option DefaultDriver: bridge
INFO[0000] Option DefaultNetwork: bridge
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
INFO[0000] Firewalld running: false
INFO[0000] Loading containers: start.
..............
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon
commit=cedd534-dirty execdriver=native-0.2 graphdriver=overlay
version=1.8.3
```

The Flannel environment for the second host can also be checked by viewing `subnet.env`:

```
# cat /run/flannel/subnet.env
FLANNEL_NETWORK=10.1.0.0/16
FLANNEL_SUBNET=10.1.31.1/24
FLANNEL_MTU=1472
FLANNEL_IPMASQ=true
```

A different subnet is allocated to the second host. The Docker service can also be restarted in this host by pointing to the Flannel bridge:

```
# source /run/flannel/subnet.env
# sudo ifconfig docker0 ${FLANNEL_SUBNET}
# sudo docker -d --bip=${FLANNEL_SUBNET} --mtu=${FLANNEL_MTU} & INFO[0000]
[graphdriver] using prior storage driver "overlay"
INFO[0000] Listening for HTTP on unix (/var/run/docker.sock)
INFO[0000] Option DefaultDriver: bridge
INFO[0000] Option DefaultNetwork: bridge
INFO[0000] Firewalld running: false
INFO[0000] Loading containers: start.
....
INFO[0000] Loading containers: done.
INFO[0000] Daemon has completed initialization
INFO[0000] Docker daemon
commit=cedd534-dirty execdriver=native-0.2 graphdriver=overlay
version=1.8.3
```

Docker containers can be created in their respective hosts, and they can be tested using the `ping` command in order to check the Flannel overlay network connectivity.

For Host 1, use the following commands:

```
#docker run -it ubuntu /bin/bash
INFO[0013] POST /v1.20/containers/create
INFO[0013] POST
/v1.20/containers/1d1582111801c8788695910e57c02fdba593f443c15e2f1db9174ed90
78db809/attach?stderr=1&stdin=1&stdout=1&stream=1
INFO[0013] POST
/v1.20/containers/1d1582111801c8788695910e57c02fdba593f443c15e2f1db9174ed90
78db809/start
INFO[0013] POST
/v1.20/containers/1d1582111801c8788695910e57c02fdba593f443c15e2f1db9174ed90
78db809/resize?h=44&w=80

root@1d1582111801:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0a:01:1e:02
          inet addr:10.1.30.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:aff:fe01:1e02/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1472  Metric:1
          RX packets:11 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:969 (969.0 B)  TX bytes:508 (508.0 B)
lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

For Host 2, use the following commands:

```
# docker run -it ubuntu /bin/bash
root@ed070166624a:/# ifconfig
eth0       Link encap:Ethernet  HWaddr 02:42:0a:01:1f:02
           inet addr:10.1.31.2  Bcast:0.0.0.0  Mask:255.255.255.0
           inet6 addr: fe80::42:aff:fe01:1f02/64 Scope:Link
           UP BROADCAST RUNNING MULTICAST  MTU:1472  Metric:1
           RX packets:18 errors:0 dropped:2 overruns:0 frame:0
           TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:0
           RX bytes:1544 (1.5 KB)  TX bytes:598 (598.0 B)
lo         Link encap:Local Loopback
           inet addr:127.0.0.1  Mask:255.0.0.0
           inet6 addr: ::1/128 Scope:Host
           UP LOOPBACK RUNNING  MTU:65536  Metric:1
           RX packets:0 errors:0 dropped:0 overruns:0 frame:0
           TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
           collisions:0 txqueuelen:0
           RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
root@ed070166624a:/# ping 10.1.30.2
PING 10.1.30.2 (10.1.30.2) 56(84) bytes of data.
64 bytes from 10.1.30.2: icmp_seq=1 ttl=60 time=3.61 ms
64 bytes from 10.1.30.2: icmp_seq=2 ttl=60 time=1.38 ms
64 bytes from 10.1.30.2: icmp_seq=3 ttl=60 time=0.695 ms
64 bytes from 10.1.30.2: icmp_seq=4 ttl=60 time=1.49 ms
```

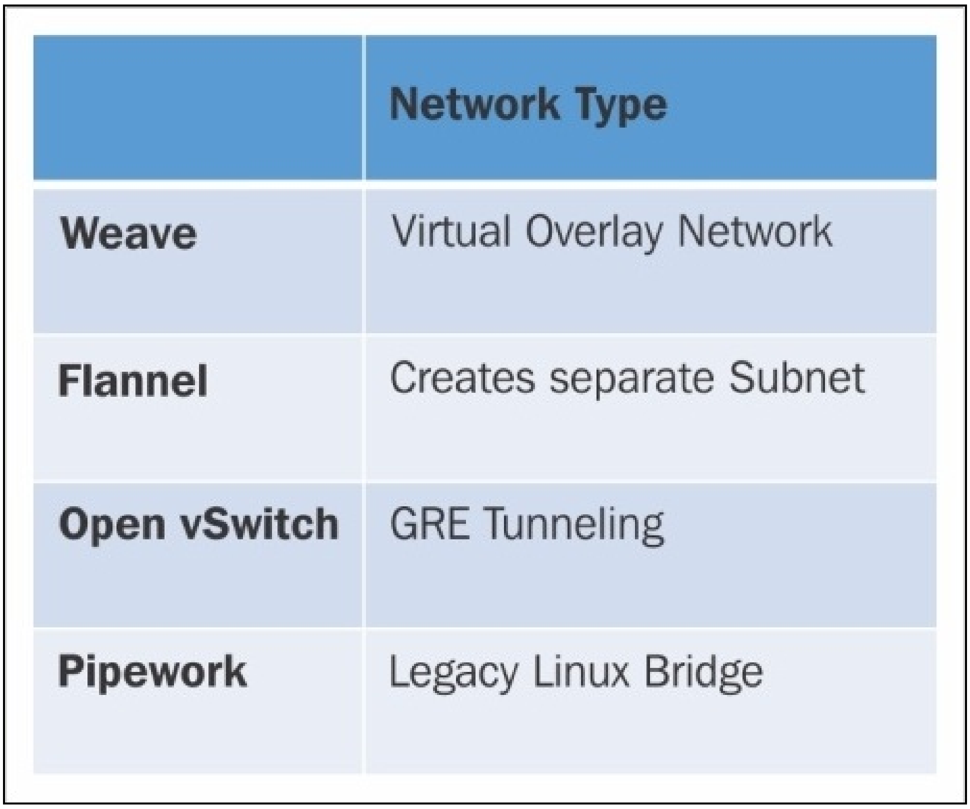
Thus, in the preceding example, we can see the complexity that Flannel reduces by running the `flanneld` agent on each host, which is responsible for allocating a subnet lease out of preconfigured address space. Flannel internally uses `etcd` to store the network configuration and other details, such as host IP and allocated subnets. The forwarding of packets is achieved using the backend strategy.

Flannel also aims to resolve the problem of Kubernetes deployment on cloud providers other than GCE, where a Flannel overlay mesh network can ease the issue of assigning a unique IP address to each pod by creating a subnet for each server.

Summary

In this chapter, we learnt how Docker containers communicate across multiple hosts using different networking options such as Weave, OVS, and Flannel. Pipework uses the legacy Linux bridge, Weave creates a virtual network, OVS uses GRE tunneling technology, and Flannel provides a separate subnet to each host in order to connect containers to multiple hosts. Some of the implementations, such as Pipework, are legacy and will become obsolete over a period of time, while others are designed to be used in the context of specific OSes, such as Flannel with CoreOS.

The following diagram shows a basic comparison of Docker networking options:



	Network Type
Weave	Virtual Overlay Network
Flannel	Creates separate Subnet
Open vSwitch	GRE Tunneling
Pipework	Legacy Linux Bridge

In the next chapter, we will discuss how Docker containers are networked when using frameworks such as Kubernetes, Docker Swarm, and Mesosphere.

Chapter 4. Networking in a Docker Cluster

In this chapter, you will learn how Docker containers are networked when using frameworks like Kubernetes, Docker Swarm, and Mesosphere.

We will cover the following topics:

- Docker Swarm
- Kubernetes
    - Networked containers in a Kubernetes cluster
    - How Kubernetes networking differs from Docker networking
    - Kubernetes on AWS
- Mesosphere

Docker Swarm

Docker Swarm is a native clustering system for Docker. Docker Swarm exposes the standard Docker API so that any tool that communicates with the Docker daemon can communicate with Docker Swarm as well. The basic aim is to allow the creation and usage of a pool of Docker hosts together. The cluster manager of Swarm schedules the containers based on the availability resources in a cluster. We can also specify the constrained resources for a container while deploying it. Swarm is designed to pack containers onto a host by saving other host resources for heavier and bigger containers rather than scheduling them randomly to a host in the cluster.

Similar to other Docker projects, Docker Swarm uses a Plug and Play architecture. Docker Swarm provides backend services to maintain a list of IP addresses in your Swarm cluster. There are several services, such as etcd, Consul, and Zookeeper; even a static file can be used. Docker Hub also provides a hosted discovery service, which is used in the normal configuration of Docker Swarm.

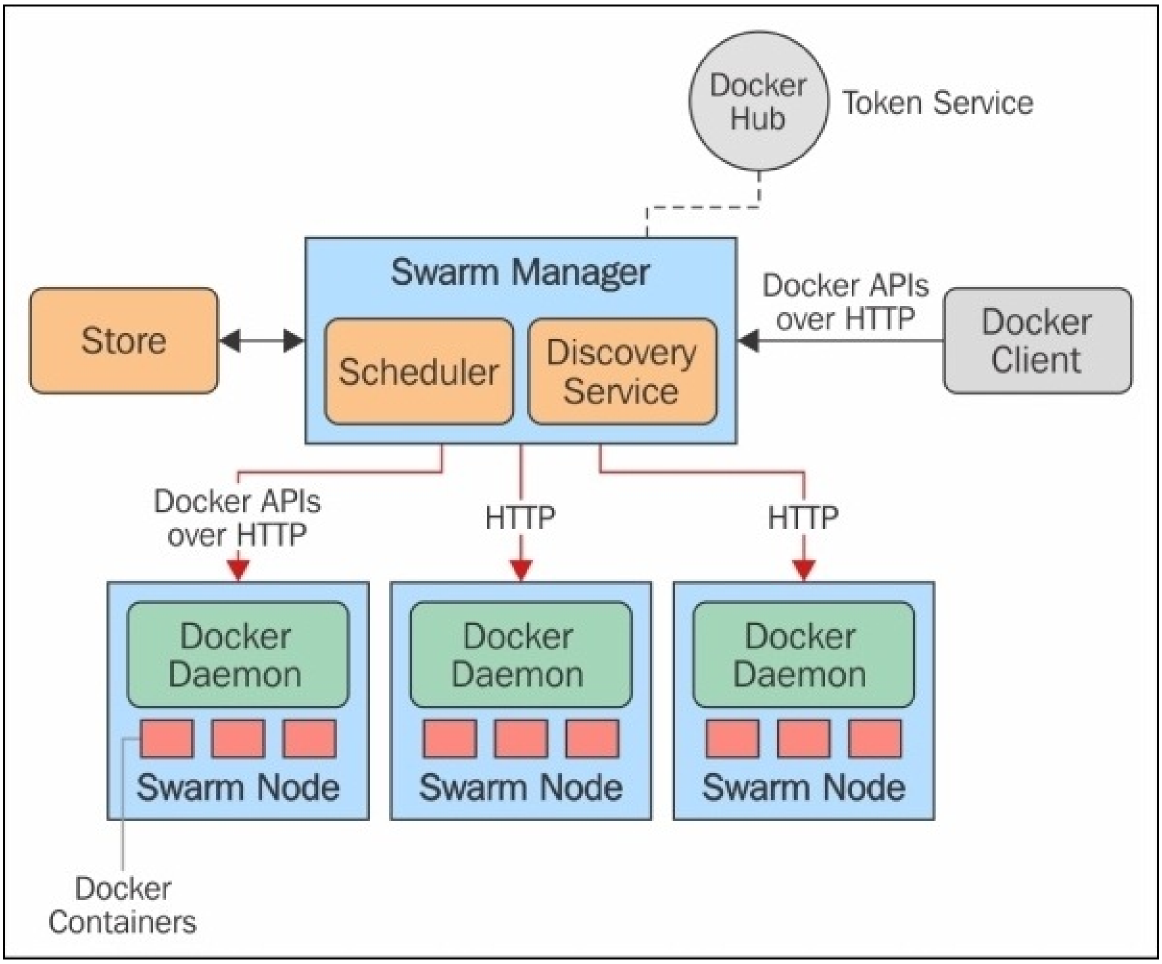
Docker Swarm scheduling uses multiple strategies in order to rank nodes. When a new container is created, Swarm places it on the node on the basis of the highest computed rank, using the following strategies:

1. **Spread**: This optimizes and schedules the containers on the nodes based on the number of containers running on the node at that point of time
2. **Binpack**: The node is selected to schedule the container on the basis of CPU and RAM utilization
3. **Random strategy**: This uses no computation; it selects the node randomly to schedule containers

Docker Swarm also uses filters in order to schedule containers, such as:

- **Constraints**: These use key/value pairs associated with nodes, such as `environment=production`
- **Affinity filter**: This is used to run a container and instruct it to locate and run next to another container based on the label, image, or identifier
- **Port filter**: In this case, the node is selected on the basis of the ports available on it
- **Dependency filter**: This co-schedules dependent containers on the same node
- **Health filter**: This prevents the scheduling of containers on unhealthy nodes

The following figure explains various components of a Docker Swarm cluster:


Docker Hub
Token Service
Swarm Manager
Store
Scheduler
Discovery Service
Docker APIs over HTTP
Docker Client
Docker APIs over HTTP
HTTP
HTTP
Docker Daemon
Swarm Node
Docker Daemon
Swarm Node
Docker Daemon
Swarm Node
Docker Containers

Docker Swarm setup

Let’s set up our Docker Swarm setup, which will have two nodes and one master.

We will be using a Docker client in order to access the Docker Swarm cluster. A Docker client can be set up on a machine or laptop and should have access to all the machines present in the Swarm cluster.

After installing Docker on all three machines, we will restart the Docker service from a command line so that it can be accessed from TCP port 2375 on the localhost (`0.0.0.0:2375`) or from a specific host IP address and can allow connections using a Unix socket on all the Swarm nodes, as follows:

```
$ docker -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock –d &
```

Docker Swarm images are required to be deployed as Docker containers on the master node. In our example, the master node’s IP address is `192.168.59.134`. Replace it with your Swarm’s master node. From the Docker client machine, we will be installing Docker Swarm on the master node using the following command:

```
$ sudo docker -H tcp://192.168.59.134:2375 run --rm swarm create
Unable to find image 'swarm' locally
Pulling repository swarm
e12f8c5e4c3b: Download complete
cf43a42a05d1: Download complete
42c4e5c90ee9: Download complete
22cf18566d05: Download complete
048068586dc5: Download complete
2ea96b3590d8: Download complete
12a239a7cb01: Download complete
26b910067c5f: Download complete
4fdfeb28bd618291eeb97a2096b3f841
```

The Swarm token generated after the execution of the command should be noted, as it will be used for the Swarm setup. In our case, it is this:

```
"4fdfeb28bd618291eeb97a2096b3f841"
```

The following are the steps to set up a two-node Docker Swarm cluster:

1. From the Docker client node, the following `docker` command is required to be executed with Node 1’s IP address (in our case, `192.168.59.135`) and the Swarm token generated in the preceding code in order to add it to the Swarm cluster:

   ```
   $ docker -H tcp://192.168.59.135:2375 run -d swarm join --
   addr=192.168.59.135:2375 token:// 4fdfeb28bd618291eeb97a2096b3f841
   Unable to find image 'swarm' locally
   Pulling repository swarm
   e12f8c5e4c3b: Download complete
   cf43a42a05d1: Download complete
   42c4e5c90ee9: Download complete
   22cf18566d05: Download complete
   048068586dc5: Download complete
   2ea96b3590d8: Download complete
   ```

```
12a239a7cb01: Download complete
26b910067c5f: Download complete
e4f268b2cc4d896431dacdafdc1bb56c98fed01f58f8154ba13908c7e6fe675b
```

2. Repeat the preceding steps for Node 2 by replacing Node 1’s IP address with Node 2’s.
3. Swarm manager is required to be set up on the master node using the following command on the Docker client node:

   ```
   $ sudo docker -H tcp://192.168.59.134:2375 run -d -p 5001:2375 swarm
   manage token:// 4fdfeb28bd618291eeb97a2096b3f841
   f06ce375758f415614dc5c6f71d5d87cf8edecffc6846cd978fe07fafc3d05d3
   ```

   The Swarm cluster is set up and can be managed using the Swarm manager residing on the master node. To list all the nodes, the following command can be executed using a Docker client:

   ```
   $ sudo docker -H tcp://192.168.59.134:2375 run --rm swarm list \
   token:// 4fdfeb28bd618291eeb97a2096b3f841
   192.168.59.135:2375
   192.168.59.136:2375
   ```

4. The following command can be used to get information about the cluster:

   ```
   $ sudo docker -H tcp://192.168.59.134:5001 info
   Containers: 0
   Strategy: spread
   Filters: affinity, health, constraint, port, dependency
   Nodes: 2
   agent-1: 192.168.59.136:2375
       └ Containers: 0
       └ Reserved CPUs: 0 / 8
       └ Reserved Memory: 0 B / 1.023 GiB
    agent-0: 192.168.59.135:2375
       └ Containers: 0
       └ Reserved CPUs: 0 / 8
       └ Reserved Memory: 0 B / 1.023 GiB
   ```

5. The test `ubuntu` container can be launched onto the cluster by specifying the name as `swarm-ubuntu` and using the following command:

   ```
   $ sudo docker -H tcp://192.168.59.134:5001 run -it --name swarm-ubuntu
   ubuntu /bin/sh
   ```

6. The container can be listed using the Swarm master’s IP address:

   ```
   $ sudo docker -H tcp://192.168.59.134:5001 ps
   ```

This completes the setup of a two-node Docker Swarm cluster.

Docker Swarm networking

Docker Swarm networking has integration with libnetwork and even provides support for overlay networks. libnetwork provides a Go implementation to connect containers; it is a robust container network model that provides network abstraction for applications and the programming interface of containers. Docker Swarm is now fully compatible with the new networking model in Docker 1.9 (note that we will be using Docker 1.9 in the following setup). The key-value store is required for overlay networks, which includes discovery, networks, IP addresses, and more information.

In the following example, we will be using Consul to understand Docker Swarm networking in a better way:

1. We will provision a VirtualBox machine called `sample-keystore` using `docker-machine`:

   ```
   $ docker-machine create -d virtualbox sample-keystore
   Running pre-create checks…
   Creating machine…
   Waiting for machine to be running, this may take a few minutes…
   Machine is running, waiting for SSH to be available…
   Detecting operating system of created instance…
   Provisioning created instance…
   Copying certs to the local machine directory…
   Copying certs to the remote machine…
   Setting Docker configuration on the remote daemon…
   To see how to connect Docker to this machine, run: docker-machine.exe
   env sample-keystore
   ```

2. We will also deploy the `progrium/consul` container on the `sample-keystore` machine on port `8500` with the following command:

   ```
   $ docker $(docker-machine config sample-keystore) run -d \
       -p "8500:8500" \
       -h "consul" \
       progrium/consul -server –bootstrap
   Unable to find image 'progrium/consul:latest' locally
   latest: Pulling from progrium/consul
   3b4d28ce80e4: Pull complete
   e5ab901dcf2d: Pull complete
   30ad296c0ea0: Pull complete
   3dba40dec256: Pull complete
   f2ef4387b95e: Pull complete
   53bc8dcc4791: Pull complete
   75ed0b50ba1d: Pull complete
   17c3a7ed5521: Pull complete
   8aca9e0ecf68: Pull complete
   4d1828359d36: Pull complete
   46ed7df7f742: Pull complete
   b5e8ce623ef8: Pull complete
   049dca6ef253: Pull complete
   bdb608bc4555: Pull complete
   8b3d489cfb73: Pull complete
   c74500bbce24: Pull complete
   ```

```
9f3e605442f6: Pull complete
d9125e9e799b: Pull complete
Digest:
sha256:8cc8023462905929df9a79ff67ee435a36848ce7a10f18d6d0faba9306b97274
Status: Downloaded newer image for progrium/consul:latest
1a1be5d207454a54137586f1211c02227215644fa0e36151b000cfcde3b0df7c
```

3. Set the local environment to the `sample-keystore` machine:

```
$ eval "$(docker-machine env sample-keystore)"
```

4. We can list the consul container as follows:

```
$ docker ps
CONTAINER ID       IMAGE           COMMAND           CREATED
STATUS        PORTS                                   NAMES
1a1be5d20745   progrium/consul  /bin/start -server  5 minutes ago  Up 5
minutes   53/tcp, 53/udp, 8300-8302/tcp, 8400/tcp, 8301-8302/udp,
0.0.0.0:8500->8500/tcp   cocky_bhaskara
```

5. Create a Swarm cluster using `docker-machine`. The two machines can be created in VirtualBox; one can act as the Swarm master. As we create each Swarm node, we will be passing the options required for Docker Engine to have an overlay network driver:

```
$ docker-machine create -d virtualbox --swarm --swarm-image="swarm" --
swarm-master --swarm-discovery="consul://$(docker-machine ip sample-
keystore):8500" --engine-opt="cluster-store=consul://$(docker-machine
ip sample-keystore):8500" --engine-opt="cluster-advertise=eth1:2376"
swarm-master
Running pre-create checks…
Creating machine…
Waiting for machine to be running, this may take a few minutes…
Machine is running, waiting for SSH to be available…
Detecting operating system of created instance…
Provisioning created instance…
Copying certs to the local machine directory…
Copying certs to the remote machine…
Setting Docker configuration on the remote daemon…
Configuring swarm…
To see how to connect Docker to this machine, run: docker-machine env
swarm-master
```

The use of the parameters used in the preceding command is as follows:

- `--swarm`: This is used to configure a machine with Swarm.
- `--engine-opt`: This option is used to define arbitrary daemon options required to be supplied. In our case, we will supply the engine daemon with the `--cluster-store` option during creation time, which tells the engine the location of the key-value store for the overlay network usability. The `--cluster-advertise` option will put the machine on the network at the specific port.
- `--swarm-discovery`: It is used to discover services to use with Swarm, in our case, `consul` will be that service.
- `--swarm-master`: This is used to configure a machine as the Swarm master.

6. Another host can also be created and added to Swarm cluster, like this:

```
$ docker-machine create -d virtualbox --swarm --swarm-
image="swarm:1.0.0-rc2" --swarm-discovery="consul://$(docker-machine ip
sample-keystore):8500" --engine-opt="cluster-store=consul://$(docker-
machine ip sample-keystore):8500" --engine-opt="cluster-
advertise=eth1:2376" swarm-node-1
Running pre-create checks…
Creating machine…
Waiting for machine to be running, this may take a few minutes…
Machine is running, waiting for SSH to be available…
Detecting operating system of created instance…
Provisioning created instance…
Copying certs to the local machine directory…
Copying certs to the remote machine…
Setting Docker configuration on the remote daemon…
Configuring swarm…
To see how to connect Docker to this machine, run: docker-machine env
swarm-node-1
```

7. The machines can be listed as follows:

```
$ docker-machine ls
NAME            ACTIVE   DRIVER       STATE     URL              SWARM
sample-keystore   -     virtualbox   Running
tcp://192.168.99.100:2376
swarm-master      -     virtualbox   Running
tcp://192.168.99.101:2376  swarm-master (master)
swarm-node-1      -     virtualbox   Running
tcp://192.168.99.102:2376   swarm-master
```

8. Now, we will set the Docker environment to `swarm-master`:

```
$ eval $(docker-machine env --swarm swarm-master)
```

9. The following command can be executed on the master in order to create the overlay network and have multihost networking:

```
$ docker network create –driver overlay sample-net
```

10. The network bridge can be checked on the master using the following command:

```
$ docker network ls
NETWORK ID         NAME           DRIVER
9f904ee27bf5      sample-net     overlay
7fca4eb8c647       bridge        bridge
b4234109be9b       none           null
cf03ee007fb4       host           host
```

11. When switching to a Swarm node, we can easily list the newly created overlay network, like this:

```
$ eval $(docker-machine env swarm-node-1)
$ docker network ls
NETWORK ID        NAME            DRIVER
7fca4eb8c647      bridge          bridge
b4234109be9b      none             null
```

```
cf03ee007fb4      host            host
9f904ee27bf5     sample-net       overlay
```

12. Once the network is created, we can start the container on any of the hosts, and it will be part of the network:

    ```
    $ eval $(docker-machine env swarm-master)
    ```

13. Start the sample `ubuntu` container with the constraint environment set to the first node:

    ```
    $ docker run -itd --name=os --net=sample-net --
    env="constraint:node==swarm-master" ubuntu
    ```

14. We can check using the `ifconfig` command that the container has two network interfaces, and it will be accessible from the container deployed using Swarm manager on any other host.

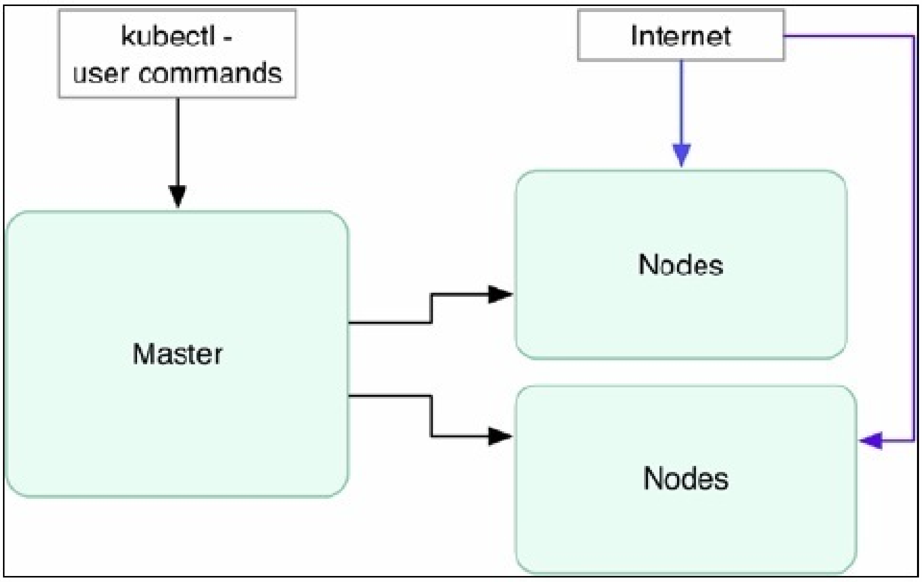
Kubernetes

Kubernetes is a container cluster management tool. Currently, it supports Docker and Rocket. It is an open source project supported by Google, and the project was launched in June 2014 at Google I/O. It supports deployment on various cloud providers such as GCE, Azure, AWS, and vSphere as well as on bare metal. The Kubernetes manager is lean, portable, extensible, and self-healing.

Kubernetes has various important components, as explained in the following list:

- **Node**: This is a physical or virtual-machine part of a Kubernetes cluster, running the Kubernetes and Docker services, onto which pods can be scheduled.
- **Master**: This maintains the runtime state of the Kubernetes server runtime. It is the point of entry for all the client calls to configure and manage Kubernetes components.
- **Kubectl**: This is the command-line tool used to interact with the Kubernetes cluster to provide master access to Kubernetes APIs. Through it, the user can deploy, delete, and list pods.
- **Pod**: This is the smallest scheduling unit in Kubernetes. It is a collection of Docker containers that share volumes and don’t have port conflicts. It can be created by defining a simple JSON file.
- **Replication controller**: It manages the lifecycle of a pod and ensures that a specified number of pods are running at a given time by creating or killing pods as required.
- **Label**: Labels are used to identify and organize pods and services based on key-value pairs.

The following diagram shows the Kubernetes Master/Minion flow:



Deploying Kubernetes on AWS

Let’s get started with Kubernetes cluster deployment on AWS, which can be done by using the config file that already exists in the Kubernetes codebase:

1. Log in to AWS Console at http://aws.amazon.com/console/.
2. Open the IAM console at https://console.aws.amazon.com/iam/home?#home.
3. Choose the IAM username, select the **Security Credentials** tab, and click on the **Create Access Key** option.
4. After the keys have been created, download and keep them in a secure place. The downloaded `.csv` file will contain an `Access Key ID` and `Secret Access Key`, which will be used to configure the AWS CLI.
5. Install and configure the AWS CLI. In this example, we have installed AWS CLI on Linux using the following command:

   ```
   $ sudo pip install awscli
   ```

6. In order to configure the AWS CLI, use the following command:

   ```
   $ aws configure
   AWS Access Key ID [None]: XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
   AWS Secret Access Key [None]: YYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYYY
   Default region name [None]: us-east-1
   Default output format [None]: text
   ```

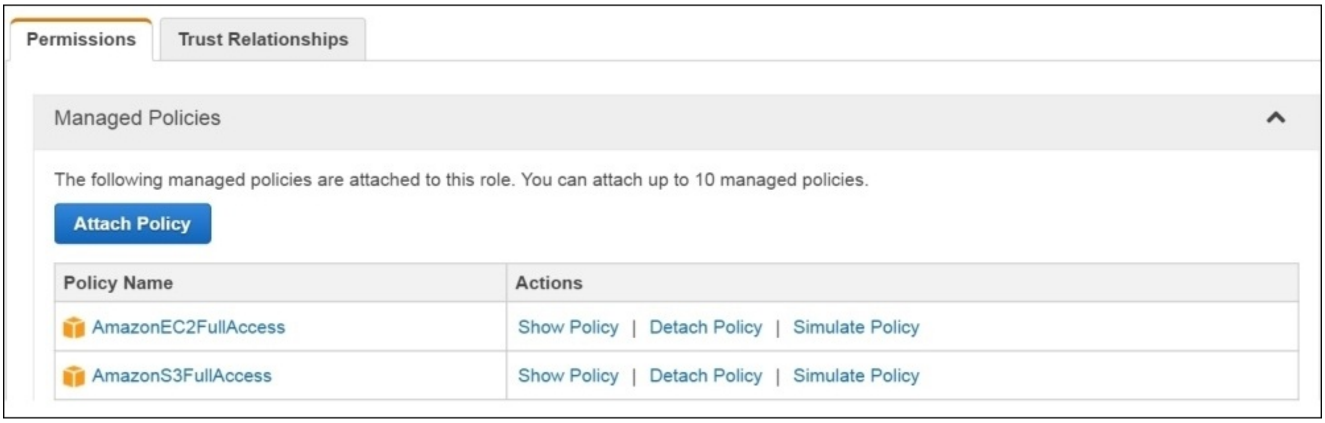
7. After configuring the AWS CLI, we will create a profile and attach a role to it with full access to S3 and EC2:

   ```
   $ aws iam create-instance-profile --instance-profile-name Kube
   ```

8. The role can be created separately using the console or AWS CLI with a JSON file that defines the permissions the role can have:

   ```
   $ aws iam create-role --role-name Test-Role --assume-role-policy-
   document /root/kubernetes/Test-Role-Trust-Policy.json
   ```

   A role can be attached to the preceding profile, which will have complete access to EC2 and S3, as shown in the following screenshot:



The following managed policies are attached to this role. You can attach up to 10 managed policies.

Policy Name	Actions
AmazonEC2FullAccess	Show Policy | Detach Policy | Simulate Policy
AmazonS3FullAccess	Show Policy | Detach Policy | Simulate Policy

9. After the creation of the role, it can be attached to a policy using the following command:

```
$ aws iam add-role-to-instance-profile --role-name Test-Role --
instance-profile-name Kube
```

10. By default, the script uses the default profile. We can change it as follows:

```
$ export AWS_DEFAULT_PROFILE=Kube
```

11. The Kubernetes cluster can be easily deployed using one command, as follows:

```
$ export KUBERNETES_PROVIDER=aws; wget -q -O - https://get.k8s.io |
bash
Downloading kubernetes release v1.1.1 to /home/vkohli/kubernetes.tar.gz
--2015-11-22 10:39:18--  https://storage.googleapis.com/kubernetes-
release/release/v1.1.1/kubernetes.tar.gz
Resolving storage.googleapis.com (storage.googleapis.com)...
216.58.220.48, 2404:6800:4007:805::2010
Connecting to storage.googleapis.com
(storage.googleapis.com)|216.58.220.48|:443… connected.
HTTP request sent, awaiting response… 200 OK
Length: 191385739 (183M) [application/x-tar]
Saving to: 'kubernetes.tar.gz'
100%[======================================>] 191,385,739 1002KB/s   in
3m 7s
2015-11-22 10:42:25 (1002 KB/s) - 'kubernetes.tar.gz' saved
[191385739/191385739]
Unpacking kubernetes release v1.1.1
Creating a kubernetes on aws…
... Starting cluster using provider: aws
... calling verify-prereqs
... calling kube-up
Starting cluster using os distro: vivid
Uploading to Amazon S3
Creating kubernetes-staging-e458a611546dc9dc0f2a2ff2322e724a
make_bucket: s3://kubernetes-staging-e458a611546dc9dc0f2a2ff2322e724a/
+++ Staging server tars to S3 Storage: kubernetes-staging-
e458a611546dc9dc0f2a2ff2322e724a/devel
upload: ../../../tmp/kubernetes.6B8Fmm/s3/kubernetes-salt.tar.gz to
s3://kubernetes-staging-
e458a611546dc9dc0f2a2ff2322e724a/devel/kubernetes-salt.tar.gz
Completed 1 of 19 part(s) with 1 file(s) remaining
```

12. The preceding command will call `kube-up.sh` and, in turn, `utils.sh` using the `config-default.sh` script, which contains the basic configuration of a K8S cluster with four nodes, as follows:

```
ZONE=${KUBE_AWS_ZONE:-us-west-2a}
MASTER_SIZE=${MASTER_SIZE:-t2.micro}
MINION_SIZE=${MINION_SIZE:-t2.micro}
NUM_MINIONS=${NUM_MINIONS:-4}
AWS_S3_REGION=${AWS_S3_REGION:-us-east-1}
```

13. The instances are `t2.micro` running Ubuntu OS. The process takes 5 to 10 minutes, after which the IP addresses of the master and minions get listed and can be used to

access the Kubernetes cluster.

Kubernetes networking and its differences to Docker networking

Kubernetes strays from the default Docker system's networking model. The objective is for each pod to have an IP at a level imparted by the system's administration namespace, which has full correspondence with other physical machines and containers over the system. Allocating IPs per pod unit makes for a clean, retrogressive, and good model where units can be dealt with much like VMs or physical hosts from the point of view of port allotment, system administration, naming, administration disclosure, burden adjustment, application design, and movement of pods from one host to another. All containers in all pods can converse with all other containers in all other pods using their addresses. This also helps move traditional applications to a container-oriented approach.

As every pod gets a real IP address, they can communicate with each other without any need for translation. By making the same configuration of IP addresses and ports both inside as well as outside of the pod, we can create a NAT-less flat address space. This is different from the standard Docker model since there, all containers have a private IP address, which will allow them to be able to access the containers on the same host. But in the case of Kubernetes, all the containers inside a pod behave as if they are on the same host and can reach each other's ports on the localhost. This reduces the isolation between containers and provides simplicity, security, and performance. Port conflict can be one of the disadvantages of this; thus, two different containers inside one pod cannot use the same port.

In GCE, using IP forwarding and advanced routing rules, each VM in a Kubernetes cluster gets an extra 256 IP addresses in order to route traffic across pods easily.

Routes in GCE allow you to implement more advanced networking functions in the VMs, such as setting up many-to-one NAT. This is leveraged by Kubernetes.

This is in addition to the main Ethernet bridge which the VM has; this bridge is termed as the container bridge `cbr0` in order to differentiate it from the Docker bridge, `docker0`. In order to transfer packets out of the GCE environment from a pod, it should undergo an SNAT to the VM's IP address, which GCE recognizes and allows.

Other implementations with the primary aim of providing an IP-per-pod model are Open vSwitch, Flannel, and Weave.

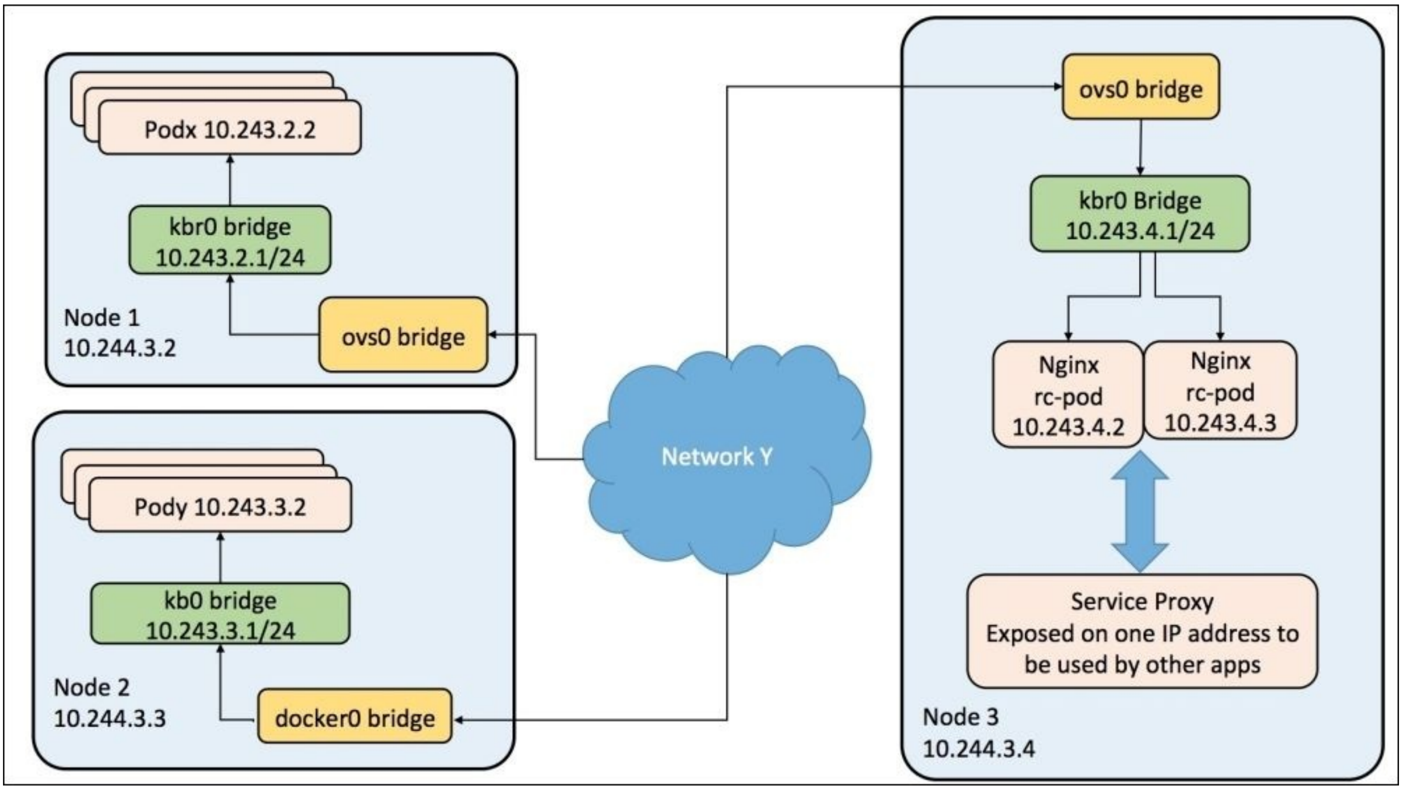
In the case of a GCE-like setup of an Open vSwitch bridge for Kubernetes, the model where the Docker bridge gets replaced by `kbr0` to provide an extra 256 subnet addresses is followed. Also, an OVS bridge (`ovs0`) is added, which adds a port to the Kubernetes bridge in order to provide GRE tunnels to transfer packets across different minions and connect pods residing on these hosts. The IP-per-pod model is also elaborated more in the upcoming diagram, where the service abstraction concept of Kubernetes is also explained.

A service is another type of abstraction that is widely used and suggested for use in Kubernetes clusters as it allows a group of pods (applications) to be accessed via virtual IP addresses and gets proxied to all internal pods in a service. An application deployed in

Kubernetes could be using three replicas of the same pod, which have different IP addresses. However, the client can still access the application on the one IP address which is exposed outside, irrespective of which backend pod takes the request. A service acts as a load balancer between different replica pods and a single point of communication for clients utilizing this application. Kubeproxy, one of the services of Kubernetes, provides load balancing and uses rules to access the service IPs and redirects them to the correct backend pod.

Deploying the Kubernetes pod

Now, in the following example, we will be deploying two nginx replication pods (`rc-pod`) and exposing them via a service in order to understand Kubernetes networking. Deciding where the application can be exposed via a virtual IP address and which replica of the pod (load balancer) the request is to be proxied to is taken care of by **Service Proxy**. Please refer to the following diagram for more details:


Podx 10.243.2.2
kbr0 bridge
10.243.2.1/24
Node 1
10.244.3.2
ovs0 bridge
Pody 10.243.3.2
kb0 bridge
10.243.3.1/24
Node 2
10.244.3.3
docker0 bridge
Network Y
ovs0 bridge
kbr0 Bridge
10.243.4.1/24
Nginx
rc-pod
10.243.4.2
Nginx
rc-pod
10.243.4.3
Service Proxy
Exposed on one IP address to
be used by other apps
Node 3
10.244.3.4

The following are the steps to deploy the Kubernetes pod:

1. In the Kubernetes master, create a new folder:

   ```
   $ mkdir nginx_kube_example
   $ cd nginx_kube_example
   ```

2. In the editor of your choice, create the `.yaml` file that will be used to deploy the nginx pods:

   ```
   $ vi nginx_pod.yaml
   ```

   Copy the following into the file:

   ```
   apiVersion: v1
   kind: ReplicationController
   metadata:
     name: nginx
   spec:
     replicas: 2
     selector:
       app: nginx
     template:
   ```

```
    metadata:
      name: nginx
      labels:
        app: nginx
  spec:
    containers:
    - name: nginx
      image: nginx
      ports:
      - containerPort: 80
```

3. Create the nginx pod using `kubectl`:

   ```
   $ kubectl create -f nginx_pod.yaml
   ```

4. In the preceding pod creation process, we created two replicas of the nginx pod, and its details can be listed using the following command:

   ```
   $ kubectl get pods
   ```

   The following is the output generated:

   ```
   NAME          READY     REASON    RESTARTS   AGE
   nginx-karne   1/1       Running   0          14s
   nginx-mo5ug   1/1       Running   0          14s
   ```

   To list replication controllers on a cluster, use the `kubectl get` command:

   ```
   $ kubectl get rc
   ```

   The following is the output generated:

   ```
   CONTROLLER   CONTAINER(S)   IMAGE(S)   SELECTOR    REPLICAS
   nginx        nginx          nginx      app=nginx   2
   ```

5. The container on the deployed minion can be listed using the following command:

   ```
   $ docker ps
   ```

   The following is the output generated:

   ```
   CONTAINER ID        IMAGE                                       COMMAND
   CREATED             STATUS              PORTS               NAMES
   1d3f9cedff1d        nginx:latest                                "nginx -g
   'daemon of   41 seconds ago      Up 40 seconds
   k8s_nginx.6171169d_nginx-karne_default_5d5bc813-3166-11e5-8256-
   ecf4bb2bbd90_886ddf56
   0b2b03b05a8d        nginx:latest                                "nginx -g
   'daemon of   41 seconds ago      Up 40 seconds
   ```

6. Deploy the nginx service using the following `.yaml` file in order to expose the nginx pod on host port `82`:

   ```
   $ vi nginx_service.yaml
   ```

   Copy the following into the file:

   ```
   apiVersion: v1
   kind: Service
   ```

```
metadata:
  labels:
    name: nginxservice
  name: nginxservice
spec:
  ports:
    # The port that this service should serve on.
    - port: 82
  # Label keys and values that must match in order to receive traffic
for this service.
  selector:
    app: nginx
  type: LoadBalancer
```

7. Create the nginx service using the `kubectl create` command:

   ```
   $kubectl create -f nginx_service.yaml
   services/nginxservice
   ```

8. The nginx service can be listed using the following command:

   ```
   $ kubectl get services
   ```

   The following is the output generated:

   ```
   NAME           LABELS                                     SELECTOR
   IP(S)          PORT(S)
   kubernetes     component=apiserver,provider=kubernetes   <none>
   192.168.3.1    443/TCP
   nginxservice   name=nginxservice                          app=nginx
   192.168.3.43   82/TCP
   ```

9. Now, the nginx server's test page can be accessed on the following URL via the service:

   `http://192.168.3.43:82`

Mesosphere

Mesosphere is a software solution that provides ways of managing server infrastructures and basically expands upon the cluster-management capabilities of Apache Mesos. Mesosphere has also launched the **DCOS** (**data center operating system**), used to manage data centers by spanning all the machines and treating them as a single computer, providing a highly scalable and elastic way of deploying apps on top of it. DCOS can be installed on any public cloud or your own private data center, ranging from AWS, GCE, and Microsoft Azure to VMware. Marathon is the framework for Mesos and is designed to launch and run applications; it serves as a replacement for the init system. Marathon provides various features such as high availability, application health check, and service discovery, which help you run applications in Mesos clustered environments.

This session describes how to bring up a single-node Mesos cluster.

Docker containers

Mesos can run and manage Docker containers using the Marathon framework.

In this exercise, we will use CentOS 7 to deploy a Mesos cluster:

1. Install Mesosphere and Marathon using the following command:

   ```
   # sudo rpm -Uvh
   http://repos.mesosphere.com/el/7/noarch/RPMS/mesosphere-el-repo-7-
   1.noarch.rpm
   # sudo yum -y install mesos marathon
   ```

   Apache Mesos uses Zookeeper to operate. Zookeeper acts as the master election service in the Mesosphere architecture and stores states for the Mesos nodes.

2. Install Zookeeper and the Zookeeper server package by pointing to the RPM repository for Zookeeper, as follows:

   ```
   # sudo rpm -Uvh http://archive.cloudera.com/cdh4/one-click-
   install/redhat/6/x86_64/cloudera-cdh-4-0.x86_64.rpm
   # sudo yum -y install zookeeper zookeeper-server
   ```

3. Validate Zookeeper by stopping and restarting it:

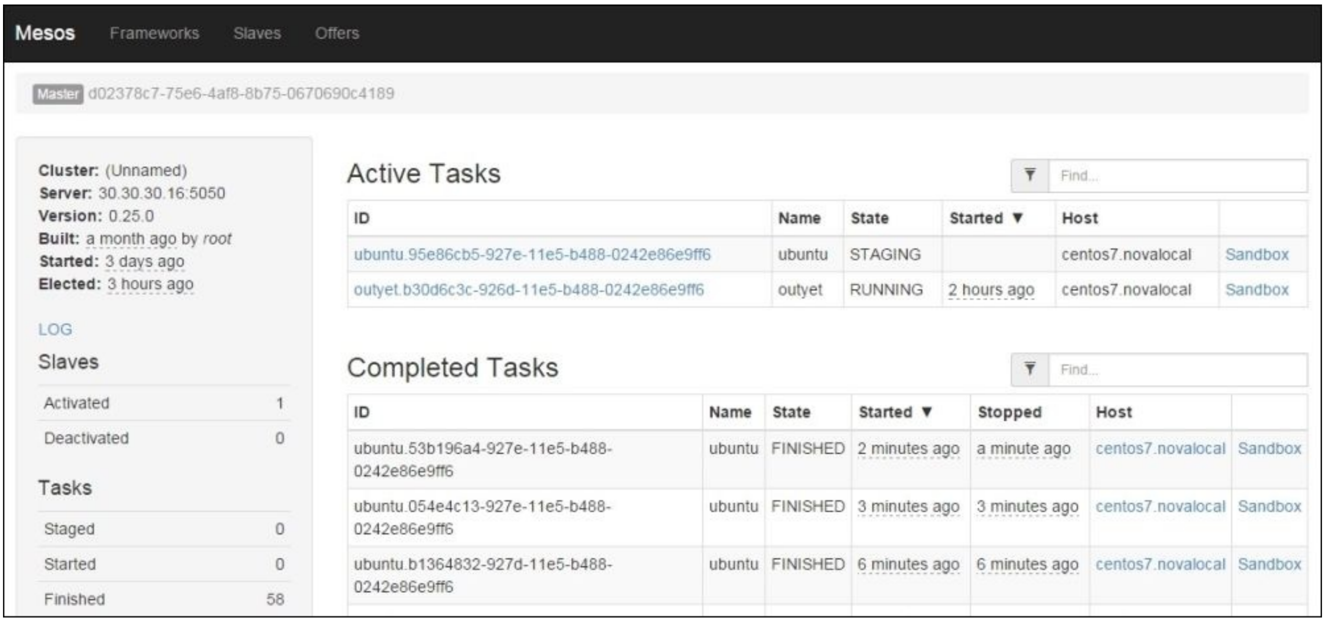
   ```
   # sudo service zookeeper-server stop
   # sudo service zookeeper-server start
   ```

   Mesos uses a simple architecture to give you intelligent task distribution across a cluster of machines without worrying about where they are scheduled.

4. Configure Apache Mesos by starting the `mesos-master` and `mesos-slave` processes as follows:

   ```
   # sudo service mesos-master start
   # sudo service mesos-slave start
   ```

5. Mesos will be running on port `5050`. As shown in the following screenshot, you can access the Mesos interface with your machine's IP address, here, `http://192.168.10.10:5050`:



Master d02378c7-75e6-4af8-8b75-0670690c4189

Cluster: (Unnamed)
Server: 30.30.30.16:5050
Version: 0.25.0
Built: a month ago by *root*
Started: 3 days ago
Elected: 3 hours ago

LOG

Slaves

Activated	1
Deactivated	0

Tasks

Staged	0
Started	0
Finished	58

Active Tasks

ID	Name	State	Started ▼	Host	
ubuntu.95e86cb5-927e-11e5-b488-0242e86e9ff6	ubuntu	STAGING		centos7.novalocal	Sandbox
outyet.b30d6c3c-926d-11e5-b488-0242e86e9ff6	outyet	RUNNING	2 hours ago	centos7.novalocal	Sandbox

Completed Tasks

ID	Name	State	Started ▼	Stopped	Host	
ubuntu.53b196a4-927e-11e5-b488-0242e86e9ff6	ubuntu	FINISHED	2 minutes ago	a minute ago	centos7.novalocal	Sandbox
ubuntu.054e4c13-927e-11e5-b488-0242e86e9ff6	ubuntu	FINISHED	3 minutes ago	3 minutes ago	centos7.novalocal	Sandbox
ubuntu.b1364832-927d-11e5-b488-0242e86e9ff6	ubuntu	FINISHED	6 minutes ago	6 minutes ago	centos7.novalocal	Sandbox

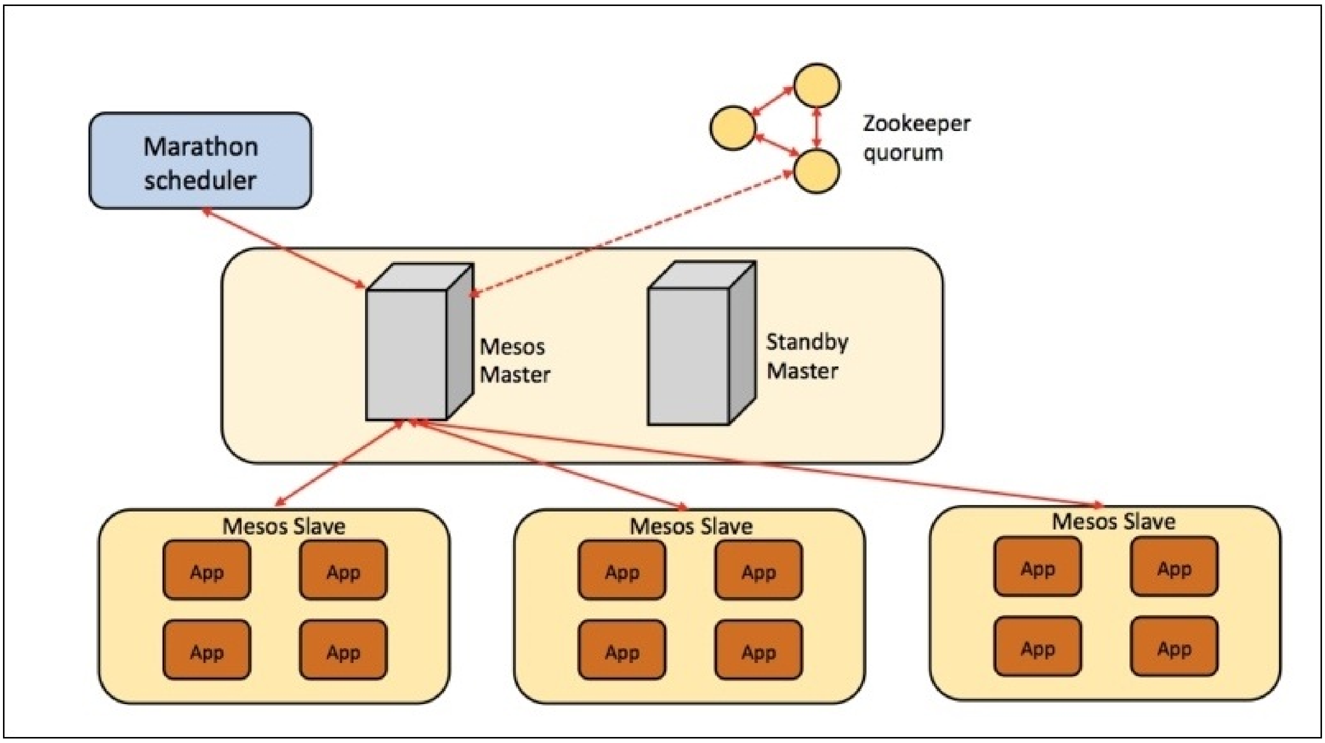
6. Test Mesos using the `mesos-execute` command:

   ```
   # export MASTER=$(mesos-resolve `cat /etc/mesos/zk` 2>/dev/null)
   # mesos help
   # mesos-execute --master=$MASTER --name="cluster-test" --command="sleep
   40"
   ```

7. With the `mesos-execute` command running, enter *Ctrl* + *Z* to suspend the command. You can see how it appears in the web UI and command line:

   ```
   # hit ctrl-z
   # mesos ps --master=$MASTER
   ```

   The Mesosphere stack uses Marathon to manage processes and services. It serves as a replacement for the traditional init system. It simplifies the running of applications in a clustered environment. The following figure shows the Mesosphere Master slave topology with Marathon:


Marathon scheduler
Zookeeper quorum
Mesos Master
Standby Master
Mesos Slave
App
App
App
App
Mesos Slave
App
App
App
App
Mesos Slave
App
App
App
App

Marathon can be used to start other Mesos frameworks; as it is designed for long-running applications, it will ensure that the applications it has launched will continue running even if the slave nodes they are running on fail.

8. Start the Marathon service using the following command:

```
# sudo service marathon start
```

You can view the Marathon GUI at `http://192.168.10.10:8080`.

Deploying a web app using Docker

In this exercise, we will install a simple Outyet web application:

1. Install Docker using the following commands:

```
# sudo yum install -y golang git device-mapper-event-libs docker
# sudo chkconfig docker on
# sudo service docker start
# export GOPATH=~/go
# go get github.com/golang/example/outyet
# cd $GOPATH/src/github.com/golang/example/outyet
# sudo docker build -t outyet.
```

2. The following command tests the Docker file before adding it to Marathon:

```
# sudo docker run --publish 6060:8080 --name test --rm outyet
```

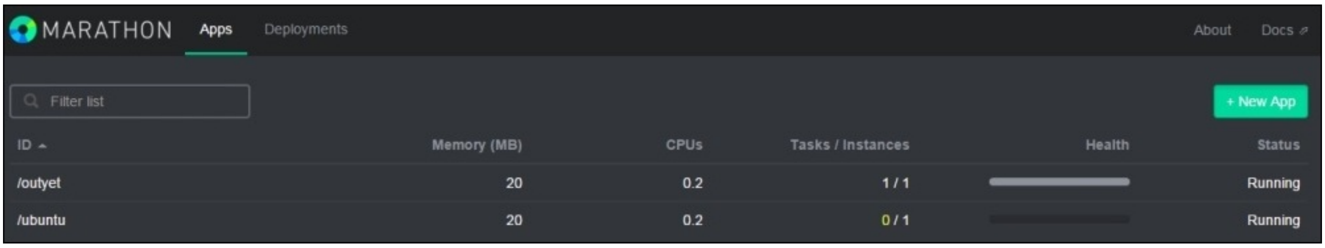
3. Go to `http://192.168.10.10:6060/` on your browser in order to confirm it works. Once it does, you can hit *CTRL* + *C* to exit the Outyet Docker.
4. Create a Marathon application using Marathon Docker support, as follows:

```
# vi /home/user/outyet.json
{
   "id": "outyet",
   "cpus": 0.2,
   "mem": 20.0,
   "instances": 1,
   "constraints": [["hostname", "UNIQUE", ""]],
   "container": {
      "type": "DOCKER",
      "docker": {
         "image": "outyet",
         "network": "BRIDGE",
         "portMappings": [ { "containerPort": 8080, "hostPort": 0,
"servicePort": 0, "protocol": "tcp" }
        ]
     }
   }
}

# echo 'docker,mesos' | sudo tee /etc/mesos-slave/containerizers
# sudo service mesos-slave restart
```

5. Containers are configured and managed better with Marathon Docker, as follows:

```
# curl -X POST http://192.168.10.10:8080/v2/apps -d
/home/user/outyet.json -H "Content-type: application/json"
```

6. You can check all your applications on the Marathon GUI at `http://192.168.10.10:8080`, as shown in the following screenshot:

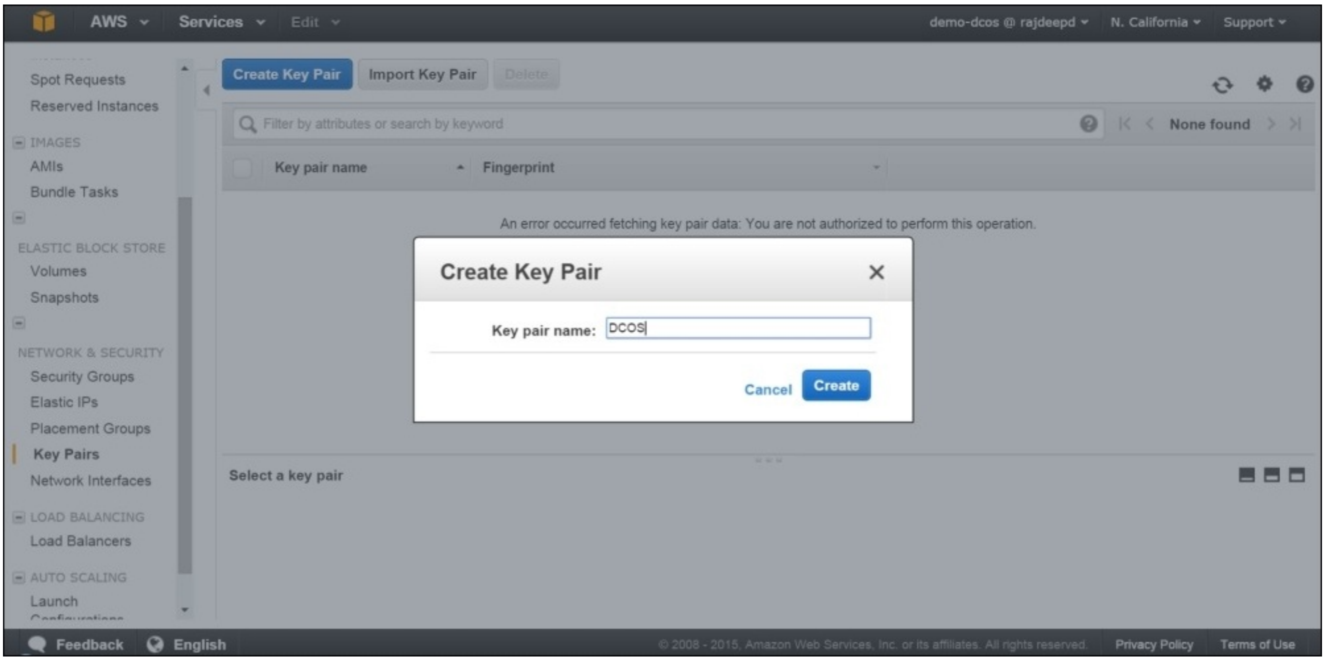

MARATHON
Apps
Deployments
About
Docs
Filter list
+ New App

ID	Memory (MB)	CPUs	Tasks / Instances	Health	Status
/outyet	20	0.2	1 / 1		Running
/ubuntu	20	0.2	0 / 1		Running

Deploying Mesos on AWS using DCOS

In this final section, we will be deploying the latest launch of DCOS by Mesosphere on AWS in order to manage and deploy Docker services in our data center:

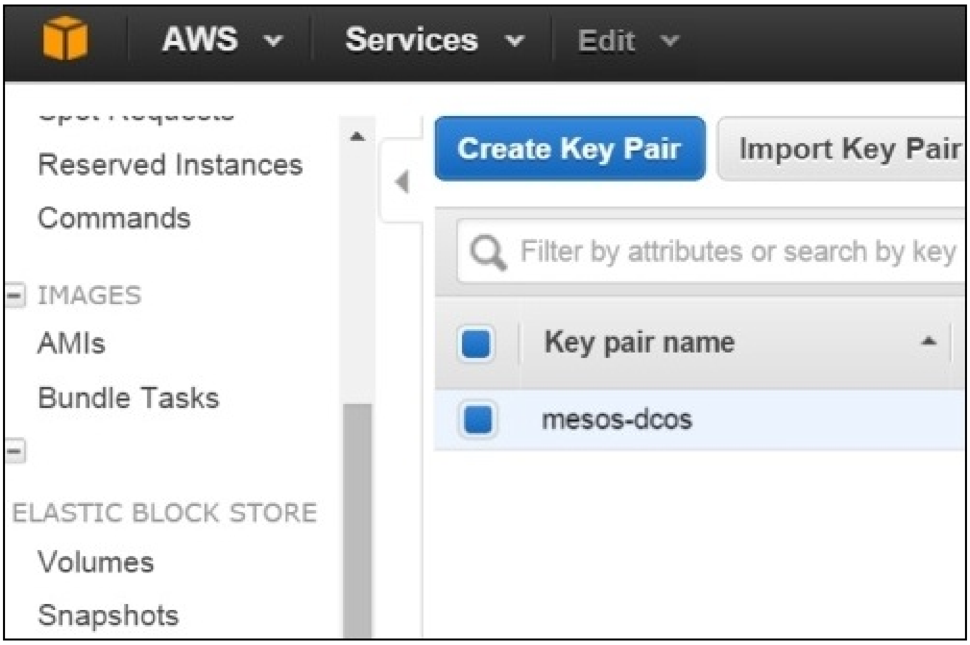
1. Create an AWS key pair in the region where the cluster is required to be deployed by going to the navigation pane and choosing **Key Pairs** under **NETWORK & SECURITY**:


AWS
Services
Edit
demo-dcos @ rajdeepd
N. California
Support
Spot Requests
Reserved Instances
IMAGES
AMIs
Bundle Tasks
ELASTIC BLOCK STORE
Volumes
Snapshots
NETWORK & SECURITY
Security Groups
Elastic IPs
Placement Groups
Key Pairs
Network Interfaces
LOAD BALANCING
Load Balancers
AUTO SCALING
Launch
Create Key Pair
Import Key Pair
Delete
Filter by attributes or search by keyword
None found
Key pair name
Fingerprint
Create Key Pair
Cancel
Create
Select a key pair
Feedback
English
© 2008 - 2015, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

An error occurred fetching key pair data: You are not authorized to perform this operation.

Key pair name: DCOS

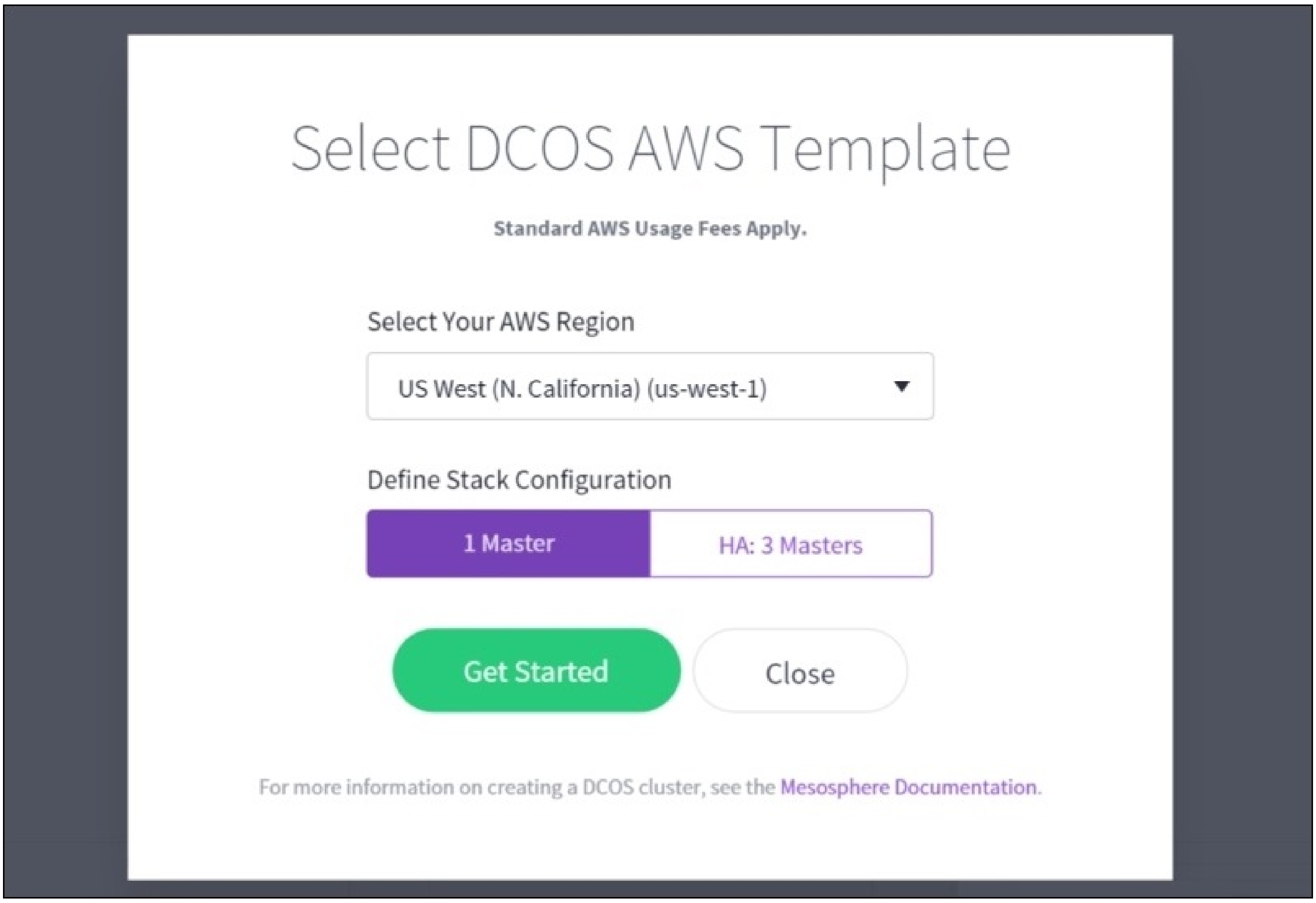
2. After being created, the key can be viewed as follows and the generated key pair (`.pem`) file should be stored in a secure location for future use:


AWS
Services
Edit
Reserved Instances
Commands
IMAGES
AMIs
Bundle Tasks
ELASTIC BLOCK STORE
Volumes
Snapshots
Create Key Pair
Import Key Pair
Filter by attributes or search by key
Key pair name

mesos-dcos

3. The DCOS cluster can be created by selecting the **1 Master** template on the official

Mesosphere site:



Select DCOS AWS Template

Standard AWS Usage Fees Apply.

Select Your AWS Region

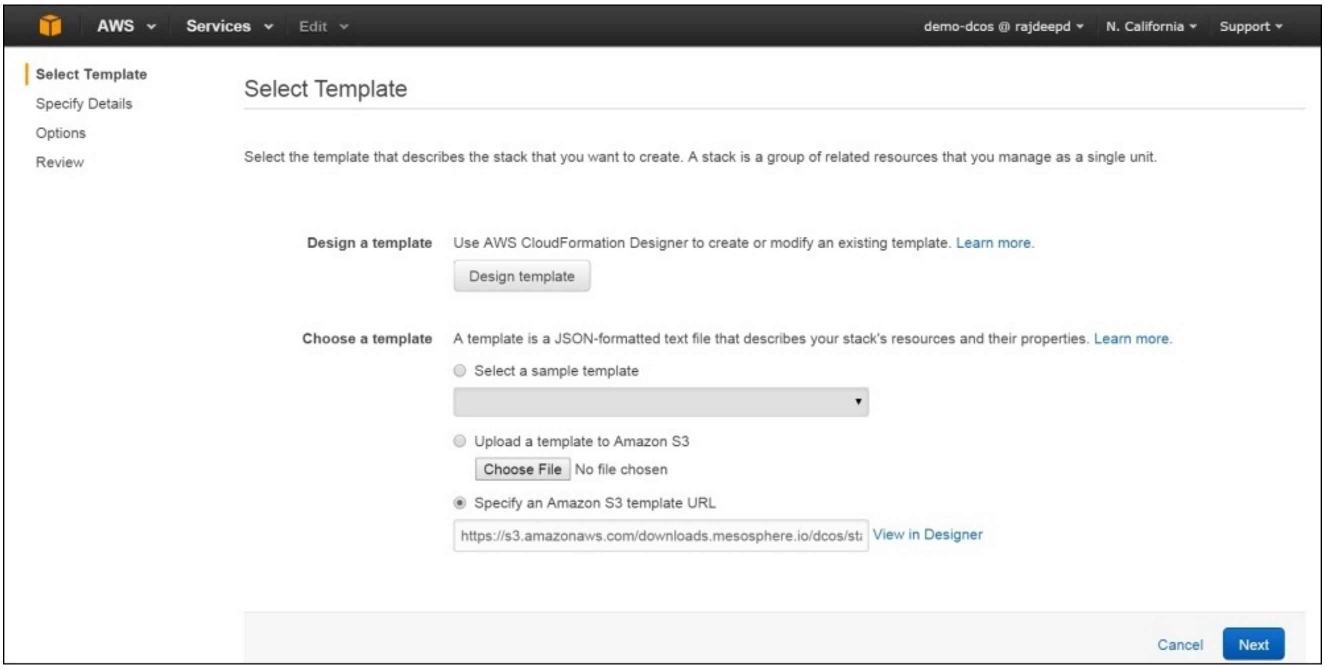
US West (N. California) (us-west-1)

Define Stack Configuration

1 Master
HA: 3 Masters
Get Started
Close

For more information on creating a DCOS cluster, see the Mesosphere Documentation.

It can also be done by providing the link for the Amazon S3 template URL in the stack deployment:



AWS
Services
Edit
demo-dcos @ rajdeepd
N. California
Support
Select Template
Specify Details
Options
Review

Select Template

Select the template that describes the stack that you want to create. A stack is a group of related resources that you manage as a single unit.

Design a template Use AWS CloudFormation Designer to create or modify an existing template. Learn more.

Design template

Choose a template A template is a JSON-formatted text file that describes your stack's resources and their properties. Learn more.

○ Select a sample template

○ Upload a template to Amazon S3

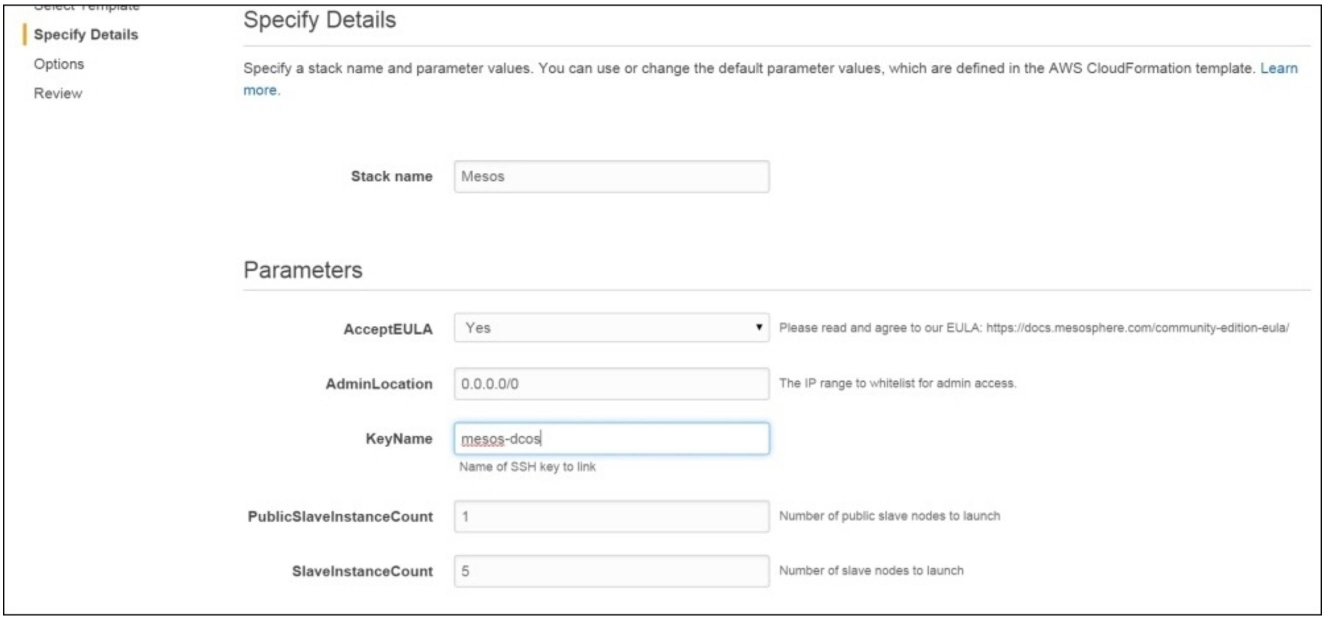
Choose File No file chosen

● Specify an Amazon S3 template URL

https://s3.amazonaws.com/downloads.mesosphere.io/dcos/st

View in Designer
Cancel
Next

4. Click on the **Next** button. Fill in the details such as **Stack name** and **KeyName**, generated in the previous step:



Specify Details

Specify a stack name and parameter values. You can use or change the default parameter values, which are defined in the AWS CloudFormation template. Learn more.

Stack name: Mesos

Parameters

AcceptEULA: Yes — Please read and agree to our EULA: https://docs.mesosphere.com/community-edition-eula/

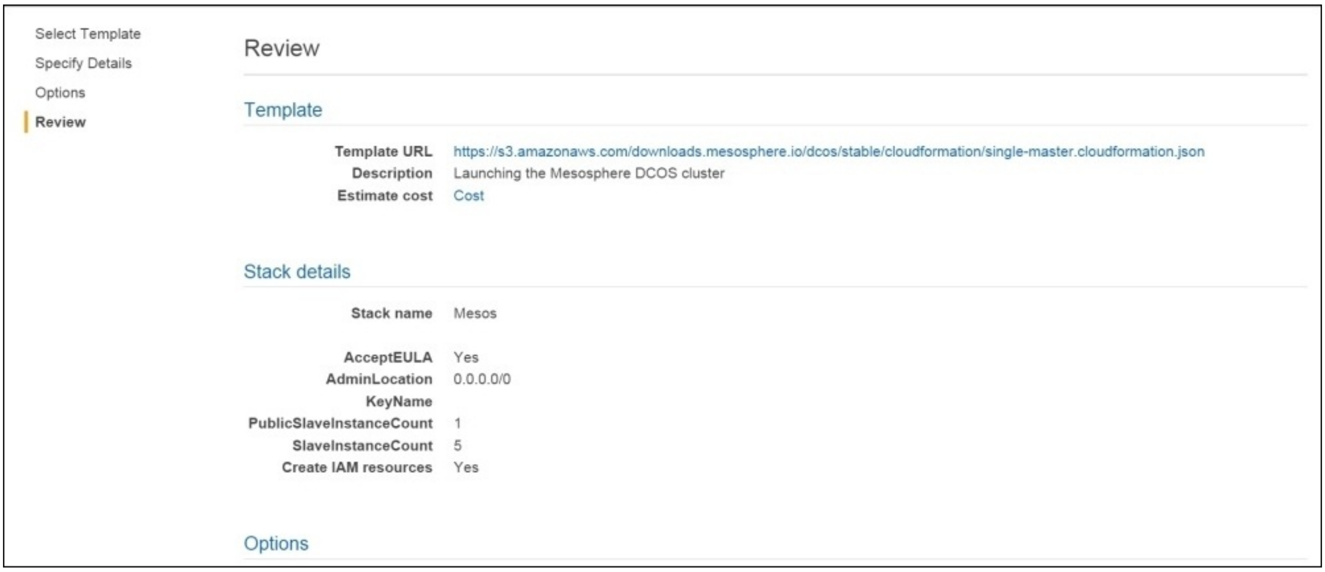
AdminLocation: 0.0.0.0/0 — The IP range to whitelist for admin access.

KeyName: mesos-dcos — Name of SSH key to link

PublicSlaveInstanceCount: 1 — Number of public slave nodes to launch

SlaveInstanceCount: 5 — Number of slave nodes to launch

5. Review the details before clicking on the **Create** button:



Review

Template

Template URL: https://s3.amazonaws.com/downloads.mesosphere.io/dcos/stable/cloudformation/single-master.cloudformation.json
Description: Launching the Mesosphere DCOS cluster
Estimate cost: Cost

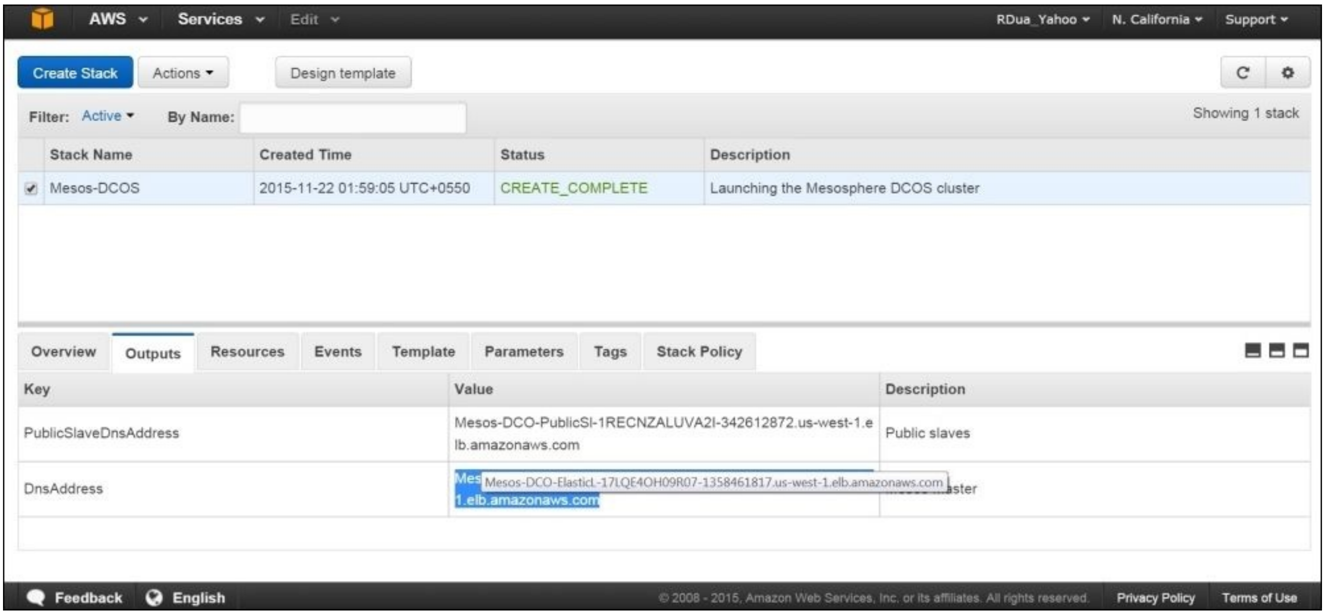
Stack details

Stack name: Mesos

AcceptEULA: Yes
AdminLocation: 0.0.0.0/0
KeyName:
PublicSlaveInstanceCount: 1
SlaveInstanceCount: 5
Create IAM resources: Yes

Options

6. After 5 to 10 minutes, the Mesos stack will be deployed and the Mesos UI can be accessed at the URL shown in the following screenshot:


AWS
Services
Edit
RDua_Yahoo
N. California
Support
Create Stack
Actions
Design template
Filter: Active
By Name:
Showing 1 stack
Overview
Outputs
Resources
Events
Template
Parameters
Tags
Stack Policy
Feedback
English
© 2008 - 2015, Amazon Web Services, Inc. or its affiliates. All rights reserved.
Privacy Policy
Terms of Use

	Stack Name	Created Time	Status	Description
☑	Mesos-DCOS	2015-11-22 01:59:05 UTC+0550	CREATE_COMPLETE	Launching the Mesosphere DCOS cluster

Key	Value	Description
PublicSlaveDnsAddress	Mesos-DCO-PublicSl-1RECNZALUVA2I-342612872.us-west-1.elb.amazonaws.com	Public slaves
DnsAddress	Mesos-DCO-ElasticL-17LQE4OH09R07-1358461817.us-west-1.elb.amazonaws.com	aster

7. Now, we will be installing the DCOS CLI on a Linux machine with Python (2.7 or 3.4) and pip preinstalled, using the following commands:

```
$ sudo pip install virtualenv
$ mkdir dcos
$ cd dcos
$ curl -O https://downloads.mesosphere.io/dcos-cli/install.sh
% Total    % Received % Xferd  Average Speed   Time    Time     Time
Current
                                 Dload  Upload   Total   Spent    Left
Speed
100  3654  100  3654    0     0   3631      0  0:00:01  0:00:01 --:--:-
-  3635
$ ls
install.sh
$ bash install.sh . http://mesos-dco-elasticl-17lqe4oh09r07-
1358461817.us-west-1.elb.amazonaws.com
Installing DCOS CLI from PyPI…
New python executable in /home/vkohli/dcos/bin/python
Installing setuptools, pip, wheel…done.
[core.reporting]: set to 'True'
[core.dcos_url]: set to 'http://mesos-dco-elasticl-17lqe4oh09r07-
1358461817.us-west-1.elb.amazonaws.com'
[core.ssl_verify]: set to 'false'
[core.timeout]: set to '5'
[package.cache]: set to '/home/vkohli/.dcos/cache'
[package.sources]: set to
'[u'https://github.com/mesosphere/universe/archive/version-1.x.zip']'
Go to the following link in your browser:
https://accounts.mesosphere.com/oauth/authorize?
scope=&redirect_uri=urn%3Aietf%3Awg%3Aoauth%3A2.0%3Aoob&response_type=c
ode&client_id=6a552732-ab9b-410d-9b7d-d8c6523b09a1&access_type=offline
Enter verification code: Skipping authentication.
Enter email address: Skipping email input.
Updating source
[https://github.com/mesosphere/universe/archive/version-1.x.zip]
```

```
Modify your bash profile to add DCOS to your PATH? [yes/no]  yes
Finished installing and configuring DCOS CLI.
Run this command to set up your environment and to get started:
source ~/.bashrc && dcos help
```

The DCOS help file can be listed as follows:

```
$ source ~/.bashrc && dcos help
Command line utility for the Mesosphere Datacenter Operating System
(DCOS). The Mesosphere DCOS is a distributed operating system built
around Apache Mesos. This utility provides tools for easy management of
a DCOS installation.
Available DCOS commands:

        config      Get and set DCOS CLI configuration properties
        help        Display command line usage information
        marathon    Deploy and manage applications on the DCOS
        node        Manage DCOS nodes
        package     Install and manage DCOS packages
        service     Manage DCOS services
        task        Manage DCOS tasks
```

8. Now, we will deploy a Spark application on top of the Mesos cluster using the DCOS package after updating it. Get a detailed command description with `dcos <command> --help`:

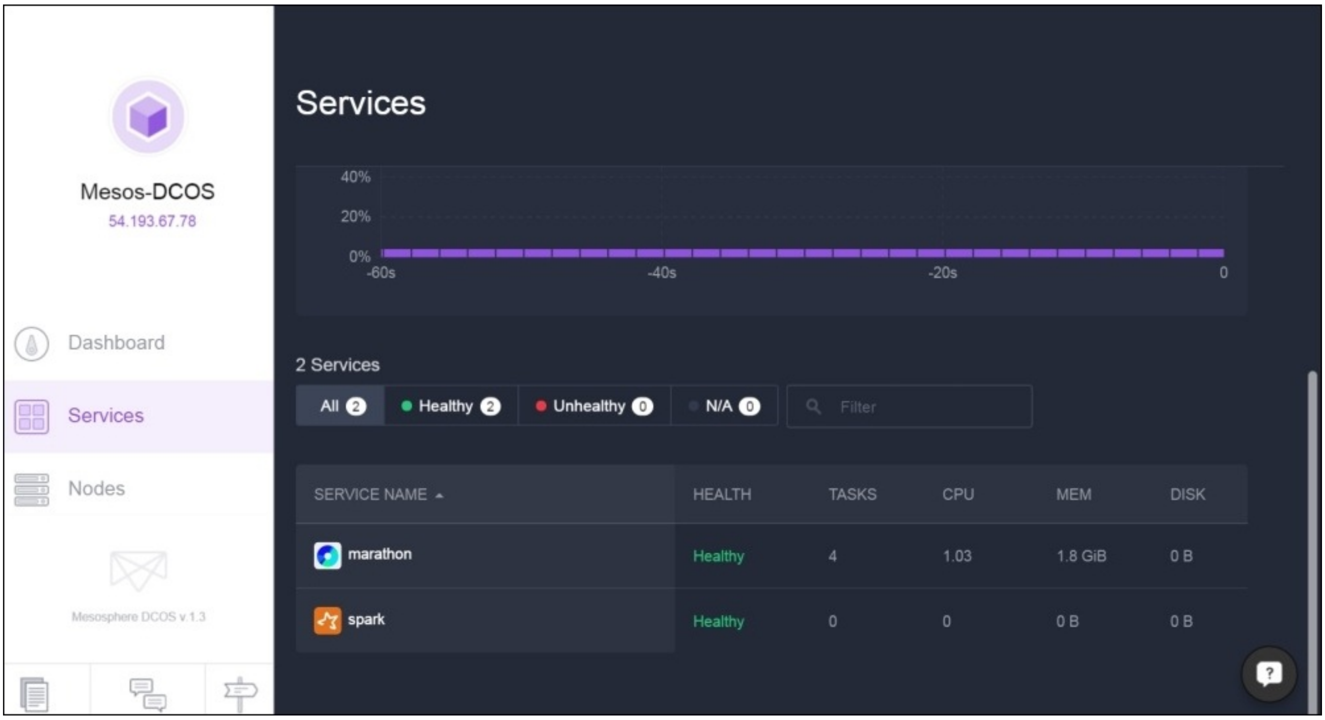
```
$ dcos config show package.sources
[
  "https://github.com/mesosphere/universe/archive/version-1.x.zip"
]
$ dcos package update
Updating source
[https://github.com/mesosphere/universe/archive/version-1.x.zip]

$ dcos package search
NAME       VERSION           FRAMEWORK     SOURCE
DESCRIPTION
arangodb   0.2.1                True
https://github.com/mesosphere/universe/archive/version-1.x.zip   A
distributed free and open-source database with a flexible data model
for documents, graphs, and key-values. Build high performance
applications using a convenient SQL-like query language or JavaScript
extensions.
cassandra  0.2.0-1              True
https://github.com/mesosphere/universe/archive/version-1.x.zip  Apache
Cassandra running on Apache Mesos.
chronos    2.4.0                True
https://github.com/mesosphere/universe/archive/version-1.x.zip  A fault
tolerant job scheduler for Mesos which handles dependencies and ISO8601
based schedules.
hdfs       0.1.7                True
https://github.com/mesosphere/universe/archive/version-1.x.zip  Hadoop
Distributed File System (HDFS), Highly Available.
kafka      0.9.2.0              True
https://github.com/mesosphere/universe/archive/version-1.x.zip  Apache
Kafka running on top of Apache Mesos.
```

```
marathon   0.11.1                 True
https://github.com/mesosphere/universe/archive/version-1.x.zip  A
cluster-wide init and control system for services in cgroups or Docker
containers.
spark      1.5.0-multi-roles-v2  True
https://github.com/mesosphere/universe/archive/version-1.x.zip  Spark
is a fast and general cluster computing system for Big Data.
```

9. The Spark package can be installed as follows:

```
$ dcos package install spark
Note that the Apache Spark DCOS Service is beta and there may be bugs,
incomplete features, incorrect documentation or other discrepancies.
We recommend a minimum of two nodes with at least 2 CPU and 2GB of RAM
available for the Spark Service and running a Spark job.
Note: The Spark CLI may take up to 5min to download depending on your
connection.
Continue installing? [yes/no] yes
Installing Marathon app for package [spark] version [1.5.0-multi-roles-
v2]
Installing CLI subcommand for package [spark] version [1.5.0-multi-
roles-v2]
```

10. After deployment, it can be seen in the DCOS UI under the **Services** tab, as shown in the following screenshot:



SERVICE NAME	HEALTH	TASKS	CPU	MEM	DISK
marathon	Healthy	4	1.03	1.8 GiB	0 B
spark	Healthy	0	0	0 B	0 B

11. In order to deploy a dummy Docker application on the preceding Marathon cluster, we can use the JSON file to define the container image, command to execute, and ports to be exposed after deployment:

```
$ nano definition.json
{
  "container": {
    "type": "DOCKER",
    "docker": {
      "image": "superguenter/demo-app"
    }
  },
  "cmd":  "python -m SimpleHTTPServer $PORT",
  "id": "demo",
  "cpus": 0.01,
  "mem": 256,
  "ports": [3000]
}
```

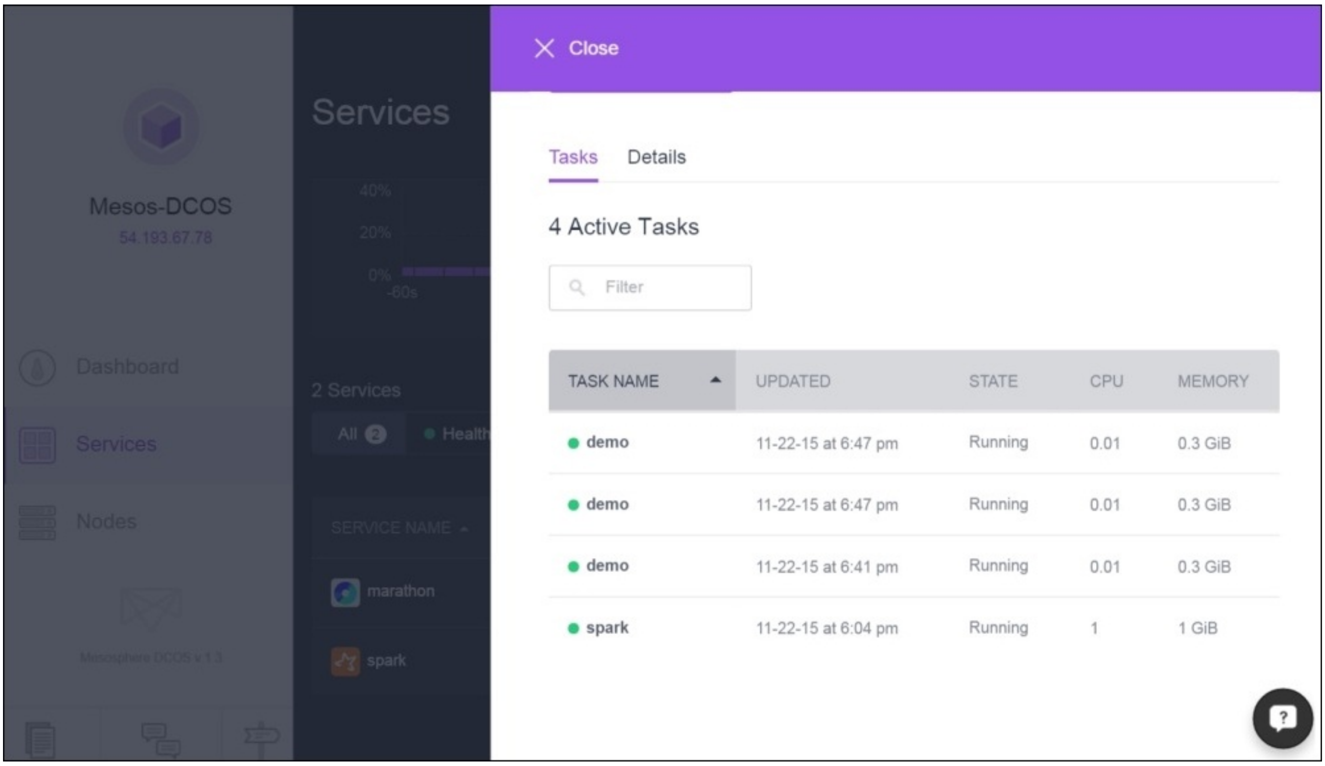
12. The app can be added to Marathon and listed as follows:

```
$ dcos marathon app add definition.json
$ dcos marathon app list
ID       MEM    CPUS  TASKS  HEALTH  DEPLOYMENT  CONTAINER  CMD
/demo   256.0   0.01   1/1    ---       ---        DOCKER   python -m
SimpleHTTPServer $PORT
/spark  1024.0  1.0    1/1    1/1       ---        DOCKER   mv
/mnt/mesos/sandbox/log4j.properties conf/log4j.properties &&
./bin/spark-class org.apache.spark.deploy.mesos.MesosClusterDispatcher
--port $PORT0 --webui-port $PORT1 --master
mesos://zk://master.mesos:2181/mesos --zk master.mesos:2181 --host
$HOST --name spark
```

13. Three instances of the preceding Docker app can be started as follows:

```
$ dcos marathon app update --force demo instances=3
Created deployment 28171707-83c2-43f7-afa1-5b66336e36d7
$ dcos marathon deployment list
APP    ACTION  PROGRESS  ID
/demo  scale     0/1     28171707-83c2-43f7-afa1-5b66336e36d7
```

14. The deployed application can be seen in the DCOS UI by clicking on the **Tasks** tab under **Services**:


Close
Tasks
Details
Filter
Mesos-DCOS
54.193.67.78
Dashboard
Services
Nodes
Mesosphere DCOS v 1.3
Services
40%
20%
0%
-60s
2 Services
All 2
Health
SERVICE NAME
marathon
spark

4 Active Tasks

TASK NAME	UPDATED	STATE	CPU	MEMORY
● **demo**	11-22-15 at 6:47 pm	Running	0.01	0.3 GiB
● **demo**	11-22-15 at 6:47 pm	Running	0.01	0.3 GiB
● **demo**	11-22-15 at 6:41 pm	Running	0.01	0.3 GiB
● **spark**	11-22-15 at 6:04 pm	Running	1	1 GiB

Summary

In this chapter, we learnt about Docker networking using various frameworks, such as the native Docker Swarm. Using libnetwork or out-of-the-box overlay networks, Swarm provides multihost networking features.

Kubernetes, on the other hand, has a different perspective from Docker, in which each pod gets its unique IP address and communication between pods can occur with the help of services. Using Open vSwitch or IP forwarding and advanced routing rules, Kubernetes networking can be enhanced to provide connectivity between pods on different subnets across hosts and the ability to expose the pods to the external world. In the case of Mesosphere, we can see that Marathon is used as the backend for the networking of the deployed containers. In the case of DCOS by Mesosphere, the entire deployed stack of machines is treated as one machine in order to provide a rich networking experience between deployed container services.

In the next chapter, we will learn about security and QoS for basic Docker networking by understanding kernel namespace, cgroups, and virtual firewalls.

Chapter 5. Security and QoS for Docker Containers

In this chapter, we will learn how security is implemented in the context of containers in general and how QoS policies are implemented to make sure that resources such as CPU and IO are shared as intended. Most of the discussion will focus on the relevance of these topics in the context of Docker.

We will cover the following in this chapter:

- File system restrictions
  - Read-only mount points
  - Copy on write
- Linux capabilities and Docker
- Securing containers in AWS ECS (EC2 container service)
- Understanding Docker security I – kernel namespaces
- Understanding Docker security II – cgroups
- Using AppArmour to secure Docker containers
- Docker security benchmark

Filesystem restrictions

In this section, we are going to study filesystem restrictions with which Docker containers are started. The following section explains the read-only mount points and copy-on-write filesystems, which are used as a base for Docker containers and the representation of kernel objects.

Read-only mount points

Docker needs access to filesystems such as sysfs and proc for processes to function. But it doesn't necessarily need to modify these mount points.

Two primary mount points loaded in read-only mode are:

- `/sys`
- `/proc`

sysfs

The sysfs filesystem is loaded into mount point `/sys`. sysfs is a mechanism for representing kernel objects, their attributes, and their relationships with each other. It provides two components:

- A kernel programming interface for exporting these items via sysfs
- A user interface to view and manipulate these items that maps back to the kernel objects that they represent

The following code shows the mount points being mounted:

```
{
  Source:      "sysfs",
  Destination: "/sys",
  Device:      "sysfs",
  Flags:       defaultMountFlags | syscall.MS_RDONLY,
},
```

A reference link for the preceding code is at https://github.com/docker/docker/blob/ecc3717cb17313186ee711e624b960b096a9334f/dae

procfs

The proc filesystem (procfs) is a special file system in Unix-like operating systems, which presents information about processes and other systems information in a hierarchical file-like structure. It is loaded into `/proc`. It provides a more convenient and standardized method for dynamically accessing process data held in the kernel than traditional tracing methods or direct access to kernel memory. It is mapped to a mount point named `/proc` at boot time:

```
{
  Source:      "proc",
  Destination: "/proc",
  Device:      "proc",
  Flags:       defaultMountFlags,
},
```

Read-only paths with `/proc`:

```
ReadonlyPaths: []string{
  "/proc/asound",
  "/proc/bus",
  "/proc/fs",
  "/proc/irq",
  "/proc/sys",
  "/proc/sysrq-trigger",
}
```

/dev/pts

This is another mount point that is mounted as read-write for the container during creation. `/dev/pts` lives purely in memory and nothing is stored on disk, hence it is safe to load it in read-write mode.

Entries in `/dev/pts` are pseudo-terminals (pty for short). Unix kernels have a generic notion of terminals. A terminal provides a way for applications to display output and to receive input through a terminal device. A process may have a controlling terminal. For a text mode application, this is how it interacts with the user:

```
{
  Source:      "devpts",
  Destination: "/dev/pts",
  Device:      "devpts",
  Flags:       syscall.MS_NOSUID | syscall.MS_NOEXEC,
  Data:        "newinstance,ptmxmode=0666,mode=0620,gid=5",
},
```

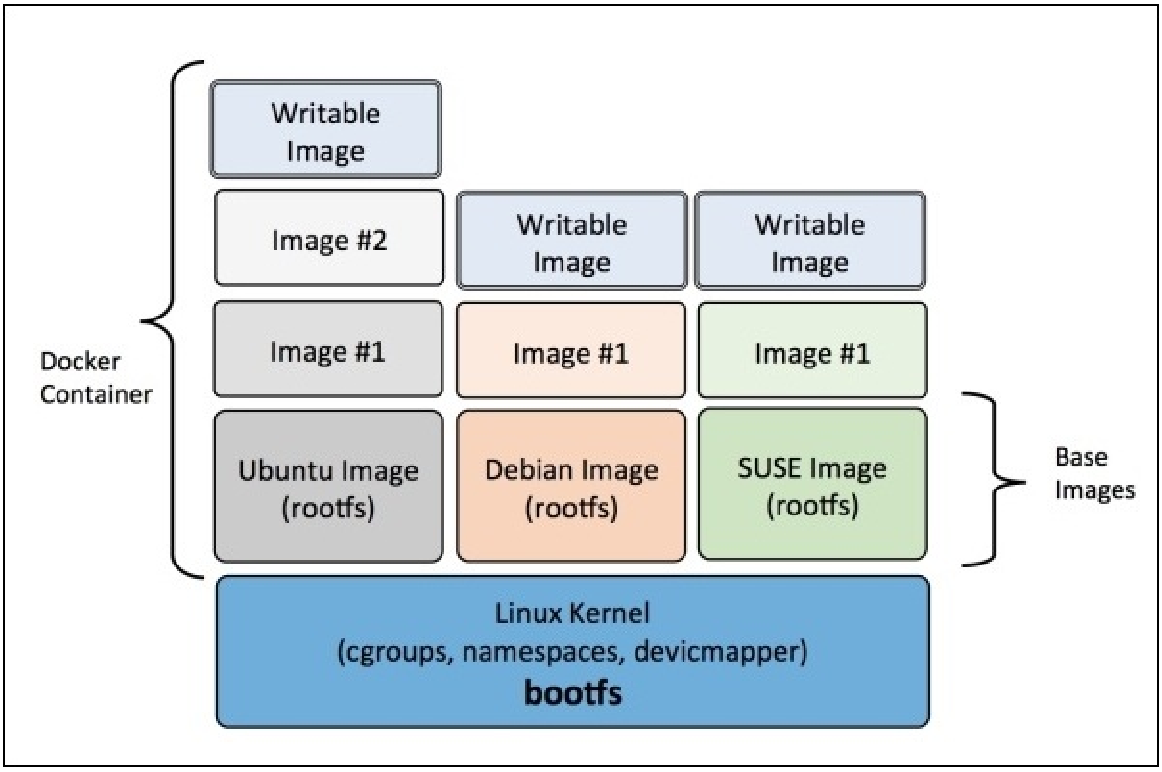
/sys/fs/cgroup

This is the mount point where cgroups are implemented and is loaded as `MS_RDONLY` for the container:

```
{
  Source:      "cgroup",
  Destination: "/sys/fs/cgroup",
  Device:      "cgroup",
  Flags:       defaultMountFlags | syscall.MS_RDONLY,
},
```

Copy-on-write

Docker uses union filesystems, which are copy-on-write filesystems. This means containers can use the same filesystem image as the base for the container. When a container writes content to the image, it gets written to a container-specific filesystem. It prevents one container from being able to access the changes of another container even if they are created from the same filesystem image. One container cannot change the image content to effect the processes in another container. The following figure explains this process:


Docker Container
Writable Image
Image #2
Writable Image
Writable Image
Image #1
Image #1
Image #1
Ubuntu Image (rootfs)
Debian Image (rootfs)
SUSE Image (rootfs)
Base Images
Linux Kernel
(cgroups, namespaces, devicmapper)
bootfs

Linux capabilities

Docker containers before 1.2 could either be given complete capabilities under privileged mode, or they can all follow a whitelist of allowed capabilities while dropping all others. If the flag `--privileged` is used, it will grant all capabilities to the container. This was not recommended for production use because it's really unsafe; it allowed Docker all privileges as a process under the direct host.

With Docker 1.2, two flags have been introduced with `docker run`:

- `--cap-add`
- `--cap-drop`

These two flags provide fine-grain control to a container, for example, as follows:

- Change the status of the Docker container's interface:

  ```
  docker run --cap-add=NET_ADMIN busybox sh -c "ip link eth0 down"
  ```

- Prevent any chown in the Docker container:

  ```
  docker run --cap-drop=CHOWN…
  ```

- Allow all capabilities except `mknod`:

  ```
  docker run --cap-add=ALL --cap-drop=MKNOD…
  ```

  Docker starts containers with a restricted set of capabilities by default. Capabilities convert a binary mode of root and non-root to a more fine-grained access control. As an example, a web server which serves HTTP request needs to be bound to port 80 for HTTP and 443 for HTTPs. These servers need not be run in the root mode. These servers can be granted `net_bind_service` capability.

  Containers and servers are a little different in this context. Servers need to run a few processes in the root mode. For example, ssh, cron, and network configurations to handle dhcp, and so on. Containers, on the other hand, do not need this access.

  The following tasks need not happen in the container:

  - ssh access is managed by Docker host
  - cron jobs should be run in the user mode
  - Network configuration such as ipconfig and routing should not happen inside the container

  We can safely deduce containers might not need root priviledges.

  Examples that can be denied are as follows:

  - Do not allow mount operations
  - Do not allow access to sockets
  - Prevent access to filesystem operations such as changing file attributes or ownership of the files
  - Prevent the container from loading new modules

Docker allows only the following capabilities:

```
Capabilities: []string{
  "CHOWN",
  "DAC_OVERRIDE",
  "FSETID",
  "FOWNER",
  "MKNOD",
  "NET_RAW",
  "SETGID",
  "SETUID",
  "SETFCAP",
  "SETPCAP",
  "NET_BIND_SERVICE",
  "SYS_CHROOT",
  "KILL",
  "AUDIT_WRITE",
},
```

A reference to the preceding code is at https://github.com/docker/docker/blob/master/daemon/execdriver/native/template/defa

A full list of available capabilities can be found in the Linux man-pages (http://man7.org/linux/man-pages/man7/capabilities.7.html).

One primary risk with running Docker containers is that the default set of capabilities and mounts given to a container may provide incomplete isolation, either independently or when used in combination with kernel vulnerabilities.

Docker supports the addition and removal of capabilities, allowing the use of a non-default profile. This may make Docker more secure through capability removal or less secure through the addition of capabilities. The best practice for users would be to remove all capabilities except those explicitly required for their processes.

Securing containers in AWS ECS

The Amazon **EC2 container service** (**ECS**) provides a highly scalable, high-performance container management service that supports Docker containers. It allows you to easily run applications on a managed cluster of Amazon EC2 instances. Amazon ECS eliminates the need for you to install, operate, and scale your own cluster management infrastructure. With simple API calls, you can launch and stop Docker-enabled applications and query the complete state of your cluster.

In the following example, we will see how to deploy a secured web application using two Docker containers, one containing a simple web application (application container), and the other containing a reverse proxy with throttling enabled (proxy container), which can be used to protect the web application. These containers will be deployed on the Amazon EC2 instance using ECS. As can be seen in the following diagram, all the network traffic will be routed through the proxy container that throttles requests. Also, we can perform activities such as filtering, logging, and intrusion detection at proxy containers using various security software.

The following are the steps to do so:

1. We will build a basic PHP web application container from the GitHub project. The following steps can be performed on a separate EC2 instance or a local machine:

   ```
   $ sudo yum install -y git
   $ git clone https://github.com/awslabs/ecs-demo-php-simple-app
   ```

2. Change directories to the `ecs-demo-php-simple-app` folder:

   ```
   $ cd ecs-demo-php-simple-app
   ```

3. We can examine `Dockerfile` as follows in order to understand the web application it will deploy:

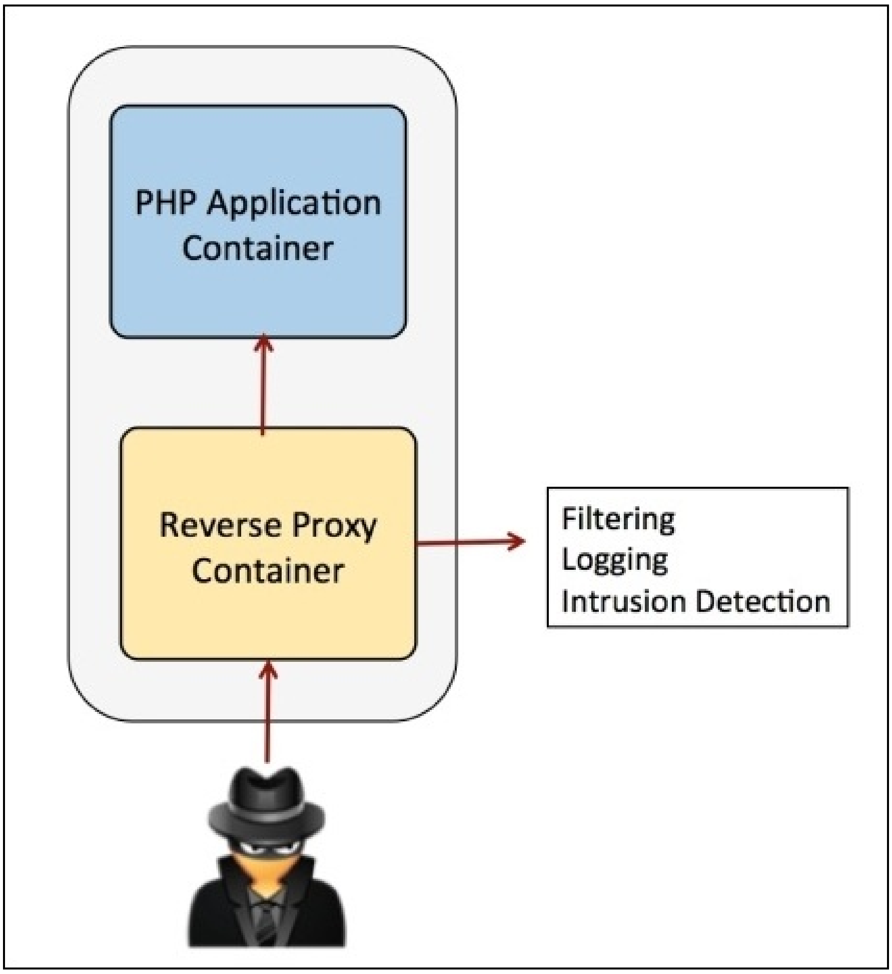
   ```
   $ cat Dockerfile
   ```

4. Build the container image using Dockerfile and then push it in your Docker Hub account. The Docker Hub account is required as it helps to deploy the containers on the Amazon ECS service by just specifying the container name:

   ```
   $ docker build -t my-dockerhub-username/amazon-ecs-sample.
   ```

   The image built over here is required to have `dockerhub-username` (correct without spaces) as the first parameter.

   The following figure depicts a hacker not able to access the web application, as the request is filtered via a proxy container and access is blocked:


PHP Application Container
Reverse Proxy Container
Filtering
Logging
Intrusion Detection

5. Upload the Docker image to the Docker Hub account:

```
$ docker login
```

6. Check to ensure your login worked:

```
$ docker info
```

7. Push your image to the Docker Hub account:

```
$ docker push my-dockerhub-username/amazon-ecs-sample
```

8. After creating the sample web application Docker container, we will now create the proxy container, which can also contain some security-related software, if required, in order to strengthen security. We will create a new proxy Docker container using a customized Dockerfile and then push the image to your Docker Hub account:

```
$ mkdir proxy-container
$ cd proxy-container
$ nano Dockerfile
FROM ubuntu
RUN apt-get update && apt-get install -y nginx
COPY nginx.conf /etc/nginx/nginx.conf
RUN echo "daemon off;" >> /etc/nginx/nginx.conf
EXPOSE 80
CMD service nginx start
```

In the previous Dockerfile we are using a base Ubuntu image and installing nginx and exposing it on port 80.

9. Next, we will create a customized `nginx.conf`, which will override the default `nginx.conf` in order to ensure the reverse proxy is configured properly:

```
user www-data;
worker_processes 4;
pid /var/run/nginx.pid;

events {
  worker_connections 768;
  # multi_accept on;
}

http {
  server {
    listen           80;

    # Proxy pass to servlet container
    location / {
      proxy_pass      http://application-container:80;
    }
  }
}
```

10. Build the proxy Docker image and push the built image to the Docker Hub account:

```
$ docker build -t my-dockerhub-username/proxy-image.
$ docker push my-dockerhub-username/proxy-image
```

11. The ECS container service can be deployed by navigating to **AWS Management Console** (https://aws.amazon.com/console/).
12. Click **Task Definitions** in the left sidebar and then click **Create a New Task Definition**.
13. Give your task definition a name, such as `SecurityApp`.
14. Next, click on **Add Container** and insert the name of the proxy web container pushed to the Docker Hub account, as well as the name of the application web container. View the contents of the JSON using **Configure via JSON** tab to see the task definition that you have created. It should be like this:

```
Proxy-container:
Container Name: proxy-container
Image: username/proxy-image
Memory: 256
Port Mappings
Host port: 80
Container port: 80
Protocol: tcp
CPU: 256
Links: application-container
Application container:
Container Name: application-container
Image: username/amazon-ecs-sample
```

```
Memory: 256
CPU: 256
```

Click the **Create** button in order to deploy the application.

15. Click **Clusters** in the left sidebar. If a default cluster does not exist, create one.
16. Launch an ECS-optimized **Amazon Machine Image** (**AMI**), ensuring it has a public IP address and a path to the Internet.
17. When your instance is up and running, navigate to the **ECS** section of the **AWS Management Console** and click **Clusters**, then **default**. Now, we should be able to see our instance under the **ECS Instances** tab.
18. Navigate to the **TASK** definitions from the left side of the **AWS Management Console** tab and click **Run Task**.
19. On the next page, ensure the cluster is set to **Default** and the number of tasks is **1**, then click **Run Task**.
20. After the process completes we can see the state of the task from a pending state to a green running state.
21. Clicking on the **ECS** tab, we can see the container instance created earlier. By clicking on it, we will get information about its public IP address. By hitting this public IP address via the browser we will be able to see our sample PHP application.

Understanding Docker security I – kernel namespaces

A namespace provides a wrapper around a global system resource of the kernel and makes the resource appear to the process within the namespace as if they have an isolated instance. Global resource changes are visible to processes in the same namespace but invisible to others. Containers are considered an excellent implementation of a kernel namespace.

The following namespaces are implemented by Docker:

- **pid namespace**: Used for process isolation (**PID—Process ID**)
- **net namespace**: Used for managing network interfaces (**NET—Networking**)
- **ipc namespace**: Used for managing access to IPC resources (**IPC—Inter Process Communication**)
- **mnt namespace**: Used for managing mount points (**MNT—Mount**)
- **uts namespace**: Used for isolating kernel and version identifiers (**UTS—Unix Time sharing System**)

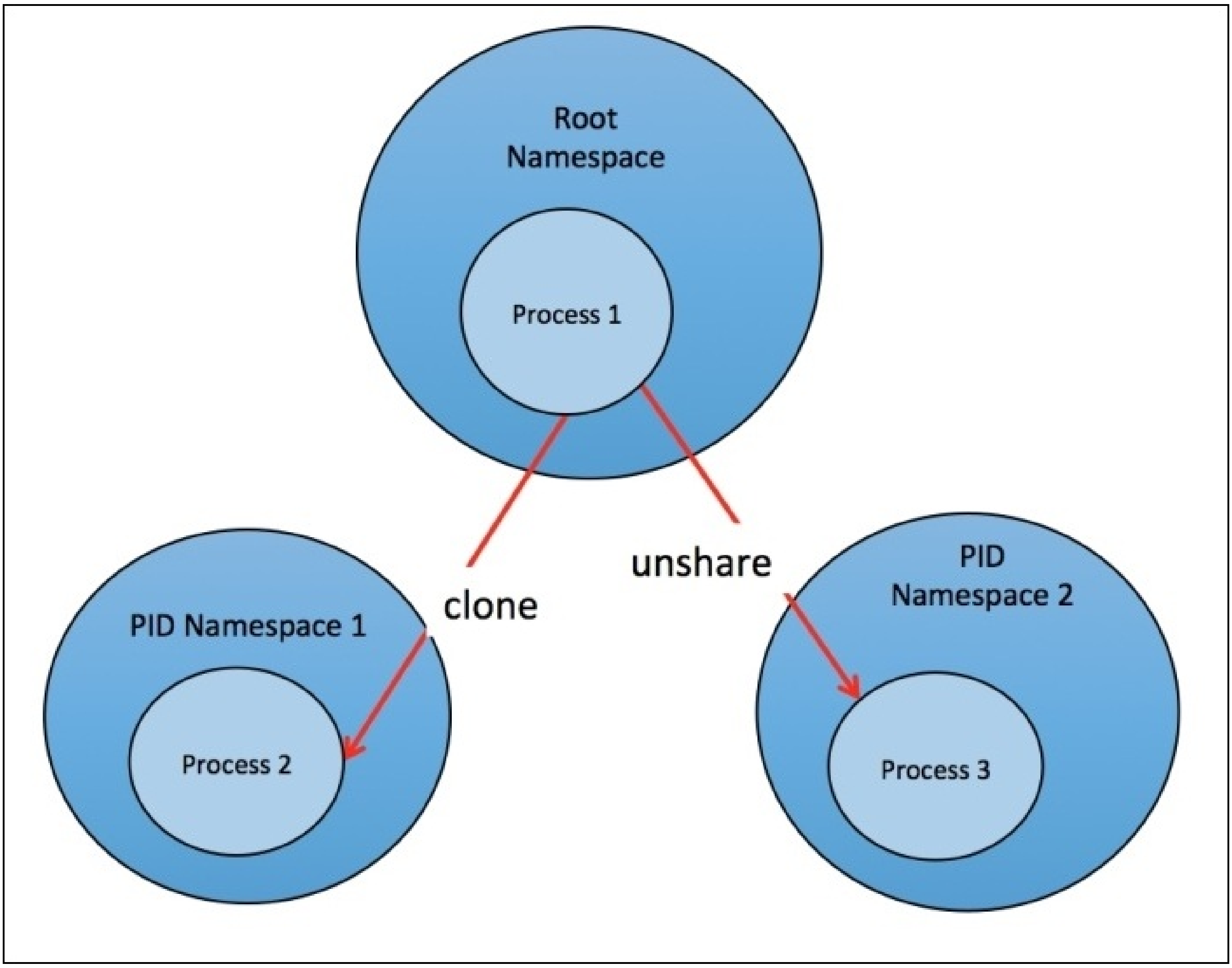
Adding namespace support in libcontainer required adding patches in the system layer of GoLang (https://codereview.appspot.com/126190043/patch/140001/150001<emphsis>src/syscall/ex so that new data structures could be maintained for PIDs, user UIDs, and so on.

pid namespace

pid namespaces isolate the process ID number space; processes in different pid namespaces can have the same pid. pid namespaces allow containers to provide functionality such as suspending/resuming the set of processes in the container, and migrating the container to a new host while the processes inside the container maintain the same pids.

pids in a new namespace start with PID 1. The kernel needs to be configured for the flag `CONFIG_PID_NS` for the namespace to work.

pid namespaces can be nested. Each pid namespace has a parent, except for the initial (root) pid namespace. The parent of a pid namespace is the pid namespace of the process that created the namespace using clone or unshare. pid namespaces form a tree, with all namespaces ultimately tracing their ancestry to the root namespace as shown in the following figure:



net namespace

net namespace provides isolation of the system resources associated with networking. Each network namespace has its own network devices, IP addresses, IP routing tables, `/proc/net` directory, port numbers, and so on.

Network namespaces make containers useful from a networking perspective: each container can have its own (virtual) network device and its own applications that bind to the per-namespace port number space; suitable routing rules in the host system can direct network packets to the network device associated with a specific container. Use of network namespaces requires a kernel that is configured with the `CONFIG_NET_NS` option (https://lwn.net/Articles/531114/).

As each container has its own network namespace, which basically means its own network interface and routing tables, net namespace is also directly leveraged by Docker to isolate IP addresses, port numbers, and so on.

Basic network namespace management

Network namespaces are created by passing a flag to the `clone()` system call, `CLONE_NEWNET`. From the command line, though, it is convenient to use the IP networking configuration tool to set up and work with network namespaces:

```
# ip netns add netns1
```

This command creates a new network namespace called `netns1`. When the IP tool creates a network namespace, it will create a bind mount for it under `/var/run/netns`, which allows the namespace to persist, even when no processes are running within it, and facilitates the manipulation of the namespace itself. Since network namespaces typically require a fair amount of configuration before they are ready for use, this feature will be appreciated by systems administrators.

The `ip netns exec` command can be used to run network management commands within the namespace:

```
# ip netns exec netns1 ip link list
1: lo: <LOOPBACK> mtu 65536 qdisc noop state DOWN mode DEFAULT
link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
```

This command lists the interfaces visible inside the namespace. A network namespace can be removed with the use of following command:

```
# ip netns delete netns1
```

This command removes the bind mount referring to the given network namespace. The namespace itself, however, will persist for as long as any processes are running within it.

Network namespace configuration

New network namespaces will have a loopback device but no other network devices. Aside from the loopback device, each network device (physical or virtual interfaces, bridges, and so on) can only be present in a single network namespace. In addition, physical devices (those connected to real hardware) cannot be assigned to namespaces other than the root. Instead, virtual network devices (for example, virtual Ethernet or vEth) can be created and assigned to a namespace. These virtual devices allow processes inside the namespace to communicate over the network; it is the configuration, routing, and so on that determines who they can communicate with.

When first created, the `lo` loopback device in the new namespace is down, so even a loopback `ping` will fail.

```
# ip netns exec netns1 ping 127.0.0.1
connect: Network is unreachable
```

In the previous command, we can see that since the net namespace for a Docker container is stored in a separate location, and thus a symlink is required to be created to `/var/run/netns`, it can be done in the following way:

```
# pid=`docker inspect -f '{{.State.Pid}}' $container_id`
# ln -s /proc/$pid/ns/net /var/run/netns/$container_id
```

In this example, it is done by bringing that interface up, which will allow the pinging of the loopback address.

```
# ip netns exec netns1 ip link set dev lo up
# ip netns exec netns1 ping 127.0.0.1
 PING 127.0.0.1 (127.0.0.1) 56(84) bytes of data.
64 bytes from 127.0.0.1: icmp_seq=1 ttl=64 time=0.052 ms
64 bytes from 127.0.0.1: icmp_seq=2 ttl=64 time=0.042 ms
64 bytes from 127.0.0.1: icmp_seq=3 ttl=64 time=0.044 ms
64 bytes from 127.0.0.1: icmp_seq=4 ttl=64 time=0.031 ms
64 bytes from 127.0.0.1: icmp_seq=5 ttl=64 time=0.042 ms
```

This still doesn’t allow communication between `netns1` and the root namespace. To do that, virtual Ethernet devices need to be created and configured.

```
# ip link add veth0 type veth peer name veth1
# ip link set veth1 netns netns1
```

The first command sets up a pair of virtual Ethernet devices that are connected. Packets sent to `veth0` will be received by `veth1` and vice versa. The second command assigns `veth1` to the `netns1` namespace.

```
# ip netns exec netns1 ifconfig veth1 10.0.0.1/24 up
# ifconfig veth0 10.0.0.2/24 up
```

Then, these two commands set IP addresses for the two devices.

```
# ping 10.0.0.1
# ip netns exec netns1 ping 10.0.0.2
```

Communication in both directions is now possible as the previous `ping` commands show.

As mentioned, though, namespaces do not share routing tables or firewall rules, as running route and `iptables -L` in `netns1` will attest:

```
# ip netns exec netns1 route
```

```
Kernel IP routing table
Destination   Gateway    Genmask        Flags    Metric Ref    Use Iface
10.0.0.0         *      255.255.255.0     U         0  0  0       veth1

# ip netns exec netns1 iptables -L
Chain INPUT (policy ACCEPT)
target     prot opt source               destination
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
Chain OUTPUT (policy ACCEPT)
target     prot opt source               destination
```

User namespace

User namespaces allows per-namespace mappings of user and group IDs. This means that user IDs and group IDs of a process inside a user namespace can be different from its IDs outside of the namespace. A process can have a non-zero user ID outside a namespace while, at the same time, having a user ID of zero inside the namespace. The process is unprivileged for operations outside the user namespace but has root privileges inside the namespace.

Creating a new user namespace

User namespaces are created by specifying the `CLONE_NEWUSER` flag when calling `clone()` or `unshare()`:

`clone()` allows the child process to share parts of its execution context with the calling process, such as the memory space, the table of file descriptors, and the table of signal handlers.

`unshare()` allows a process (or thread) to disassociate parts of its execution context that are currently being shared with other processes (or threads). Part of the execution context, such as the mount namespace, is shared implicitly when a new process is created using `fork()` or `vfork()`.

As mentioned previously, Docker containers are very similar to LXC containers as a set of namespaces and control groups are created separately for containers. Each container gets its own network stack and namespace. Until and unless containers do not have the privileged access, they are not allowed to access other hosts sockets or interfaces. If the host network mode is given to the container, then only it gets the ability to access the host ports and IP address, which can cause a potential threat to other programs running on the host.

As shown in the following example, where we use the `host` network mode in the container and it is able to access all the hosts bridges:

```
docker run -it --net=host ubuntu /bin/bash
$ ifconfig
docker0   Link encap:Ethernet  HWaddr 02:42:1d:36:0d:0d
          inet addr:172.17.0.1  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:1dff:fe36:d0d/64 Scope:Link
          UP BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:24 errors:0 dropped:0 overruns:0 frame:0
          TX packets:38 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1608 (1.6 KB)  TX bytes:5800 (5.8 KB)

eno16777736 Link encap:Ethernet  HWaddr 00:0c:29:02:b9:13
          inet addr:192.168.218.129  Bcast:192.168.218.255
Mask:255.255.255.0
          inet6 addr: fe80::20c:29ff:fe02:b913/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4934 errors:0 dropped:0 overruns:0 frame:0
          TX packets:4544 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:2909561 (2.9 MB)  TX bytes:577079 (577.0 KB)
```

```
$ docker ps -q | xargs docker inspect --format '{{ .Id }}: NetworkMode={{
.HostConfig.NetworkMode }}'
52afb14d08b9271bd96045bebd508325a2adff98dbef8c10c63294989441954d:
NetworkMode=host
```

While auditing, it should be checked that all the containers, by default, have network mode set to `default` and not `host`:

```
$ docker ps -q | xargs docker inspect --format '{{ .Id }}: NetworkMode={{
.HostConfig.NetworkMode }}'
1aca7fe47882da0952702c383815fc650f24da2c94029b5ad8af165239b78968:
NetworkMode=default
```

Each Docker container is connected to an Ethernet bridge in order to provide inter-connectivity between containers. They can ping each other to send/receive UDP packets and establish TCP connections, but that can be restricted if necessary. Namespace also provides a straightforward isolation in restricting the access of the processes running in the other container as well as the host.

We will be using the following `nsenter` command line utility in order to enter into namespaces. It is an open-source project on GitHub available at https://github.com/jpetazzo/nsenter.

Using it, we will try to enter existing container namespaces or try to spawn a new set of namespaces. It is different from the Docker `exec` command as `nsenter` doesn't enter the cgroups, which gives potential benefits for debugging and external audits by escaping the resource limitations using namespace.

We can install `nsenter` from PyPI (it requires Python 3.4) and use the command line utility to connect to a running container:

```
$ pip install nsenter
```

To replace pid with the container's pid, use the following command:

```
$ sudo nsenter --net --target=PID /bin/ip a
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN group
default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
14: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group
default
    link/ether 02:42:ac:11:00:06 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.6/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe11:6/64 scope link
       valid_lft forever preferred_lft forever
```

We can use the `docker inspect` command to make it more convenient as follows:

1. First start a new nginx server:

   ```
   $ docker run -d --name=nginx -t nginx
   ```

2. Then get pid of the container:

   ```
   PID=$(docker inspect --format {{.State.Pid}} nginx)
   ```

3. Connect to the running nginx container:

   ```
   $ nsenter --target $PID --uts --ipc --net –pid
   ```

`docker-enter` is also one of the wrappers that can be used to enter inside the container specifying the shell commands, and if no command is specified, a shell will be invoked instead. If it is required to inspect or manipulate containers without executing another command line tool, we can use context manager to do this:

```
import subprocess
from nsenter import Namespace
with Namespace(mypid, 'net'):
# output network interfaces as seen from within the mypid's net NS:
  subprocess.check_output(['ip', 'a'])
```

Understanding Docker security II – cgroups

In this section, we look at how cgroups form the backbone of isolation for a container.

Defining cgroups

Control groups provide a mechanism for aggregating/partitioning sets of tasks (processes), and all their future children, into hierarchical groups.

A cgroup associates a set a tasks with parameters from a subsystem. A subsystem itself is a resource controller used to define boundaries for cgroups or for provisioning a resource.

A hierarchy is a set of cgroups arranged in a tree, such that every task in the system is in exactly one of the cgroups in the hierarchy and a set of subsystems.

Why are cgroups required?

There are multiple efforts to provide process aggregations in the Linux kernel, mainly for resource-tracking purposes.

Such efforts include cpusets, CKRM/ResGroups, UserBeanCounters, and virtual server namespaces. These all require the basic notion of a grouping/partitioning of processes, with newly forked processes ending up in the same group (cgroup) as their parent process.

The kernel cgroup patch provides essential kernel mechanisms to efficiently implement such groups. It has minimal impact on the system fast paths and provides hooks for specific subsystems such as cpusets to provide additional behavior as desired.

Creating a cgroup manually

In the following steps, we will create a `cpuset` control group:

```
# mount -t tmpfs cgroup_root /sys/fs/cgroup
```

`tmpfs` is a file system that keeps all files in virtual memory. Everything in `tmpfs` is temporary in the sense that no files will be created on your hard drive. If you unmount a `tmpfs` instance, everything stored therein is lost:

```
# mkdir /sys/fs/cgroup/cpuset
# mount -t cgroup -ocpuset cpuset /sys/fs/cgroup/cpuset
# cd /sys/fs/cgroup/cpuset
# mkdir Charlie
# cd Charlie
# ls
cgroup.clone_children  cpuset.cpu_exclusive  cpuset.mem_hardwall
cpuset.memory_spread_page  cpuset.sched_load_balance  tasks
cgroup.event_control   cpuset.cpus           cpuset.memory_migrate
cpuset.memory_spread_slab  cpuset.sched_relax_domain_level
cgroup.procs           cpuset.mem_exclusive  cpuset.memory_pressure
cpuset.mems                notify_on_release
```

Assign CPU and memory limits to this cgroup:

```
# /bin/echo 2-3 > cpuset.cpus
# /bin/echo 0 > cpuset.mems
# /bin/echo $$ > tasks
```

The following command shows `/Charlie` as the cpuset cgroup:

```
# cat /proc/self/cgroup
11:name=systemd:/user/1000.user/c2.session
10:hugetlb:/user/1000.user/c2.session
9:perf_event:/user/1000.user/c2.session
8:blkio:/user/1000.user/c2.session
7:freezer:/user/1000.user/c2.session
6:devices:/user/1000.user/c2.session
5:memory:/user/1000.user/c2.session
4:cpuacct:/user/1000.user/c2.session
3:cpu:/user/1000.user/c2.session
2:cpuset:/Charlie
```

Attaching processes to cgroups

Add the process ID `PID{X}` to the tasks file as shown in the following:

```
# /bin/echo PID > tasks
```

Note that it is `PID`, not PIDs.

You can only attach one task at a time. If you have several tasks to attach, you have to do it one after another:

```
# /bin/echo PID1 > tasks
# /bin/echo PID2 > tasks
...
# /bin/echo PIDn > tasks
```

Attach the current shell task by echoing `0`:

```
# echo 0 > tasks
```

Docker and cgroups

cgroups are managed as part of the libcontainer project under Docker's GitHub repo (https://github.com/opencontainers/runc/tree/master/libcontainer/cgroups). There is a cgroup manager that manages the interaction with the cgroup APIs in the kernel.

The following code shows the lifecycle events managed by the manager:

```
type Manager interface {
  // Apply cgroup configuration to the process with the specified pid
  Apply(pid int) error
  // Returns the PIDs inside the cgroup set
  GetPids() ([]int, error)
  // Returns statistics for the cgroup set
  GetStats() (*Stats, error)
  // Toggles the freezer cgroup according with specified state
  Freeze(state configs.FreezerState) error
  // Destroys the cgroup set
  Destroy() error
  // Paths maps cgroup subsystem to path at which it is mounted.
  // Cgroups specifies specific cgroup settings for the various subsystems
  // Returns cgroup paths to save in a state file and to be able to
  // restore the object later.
  GetPaths() map[string]string
  // Set the cgroup as configured.
  Set(container *configs.Config) error
}
```

Using AppArmor to secure Docker containers

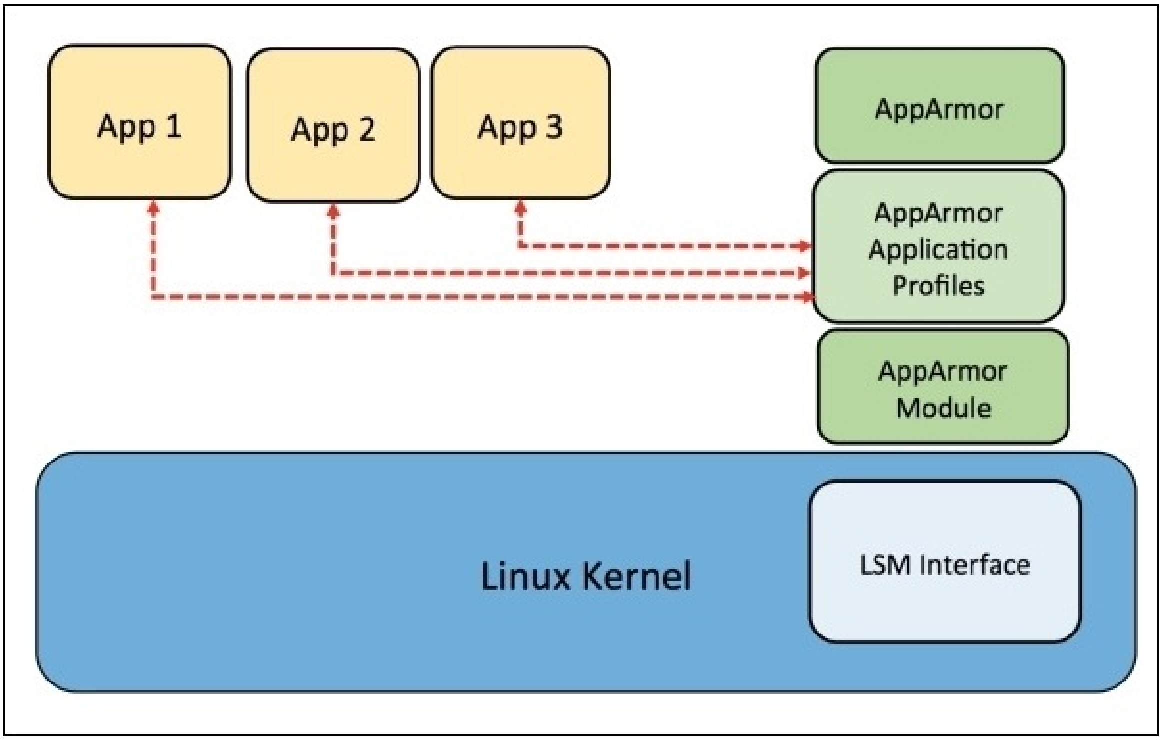
AppArmor is a **Mandatory Access Control** (**MAC**) system that is a kernel enhancement to confine programs to a limited set of resources. AppArmor's security model is to bind access control attributes to programs rather than to users.

AppArmor confinement is provided via profiles loaded into the kernel, typically on boot. AppArmor profiles can be in one of two modes: enforcement or complain.

Profiles loaded in enforcement mode will result in enforcement of the policy defined in the profile, as well as reporting policy violation attempts (either via syslog or auditd).

Profiles in complain mode will not enforce policy but instead report policy violation attempts.

AppArmor differs from some other MAC systems on Linux: it is path-based, it allows mixing of enforcement and complain-mode profiles, it uses include files to ease development, and it has a far lower barrier to entry than other popular MAC systems. The following figure shows the AppArmour application profiles linked to apps:



AppArmor is an established technology first seen in Immunix and later integrated into Ubuntu, Novell/SUSE, and Mandriva. Core AppArmor functionality is in the mainline Linux kernel from 2.6.36 onwards; work is ongoing by AppArmor, Ubuntu, and other developers to merge additional AppArmor functionality into the mainline kernel.

You can find more information about AppArmor at https://wiki.ubuntu.com/AppArmor.

AppArmor and Docker

Applications running inside Docker can leverage AppArmor for defining policies. These profiles can either be created manually or loaded using a tool called bane.

Note

On Ubuntu 14.x, make sure systemd is installed for the following commands to work.

The following steps show how to use this tool:

1. Download the bane project for GitHub:

   ```
   $ git clone https://github.com/jfrazelle/bane
   ```

   Make sure this is done in the directory in your GOPATH. For example, we used `/home/ubuntu/go` and the bane source was downloaded in `/home/Ubuntu/go/src/github.com/jfrazelle/bane`.

2. Install toml parser needed by bane to be compiled:

   ```
   $ go get github.com/BurntSushi/toml
   ```

3. Go to the `/home/Ubuntu/go/src/github.com/jfrazelle/bane` directory and run the following command:

   ```
   $ go install
   ```

4. You will find the bane binary in `/home/Ubuntu/go/bin`.
5. Use a `.toml` file to create a profile:

   ```
   Name = "nginx-sample"
   [Filesystem]
   # read only paths for the container
   ReadOnlyPaths = [
     "/bin/**",
     "/boot/**",
     "/dev/**",
     "/etc/**",
     …
   ]
   AllowExec = [
     "/usr/sbin/nginx"
   ]
   # denied executable files
   DenyExec = [
     "/bin/dash",
     "/bin/sh",
     "/usr/bin/top"
   ]
   ```

6. Execute bane to load the profile. `sample.toml` is a file in the directory `/home/Ubuntu/go/src/github.com/jfrazelle/bane`:

   ```
   $ sudo bane sample.toml
   # Profile installed successfully you can now run the profile with #
   ```

```
`docker run --security-opt="apparmor:docker-nginx-sample"`
```

This profile will make a whole lot of paths read only and allows only nginx execution in the container we are going to create. It disables TOP, PING, and so on.

7. Once the profile is loaded you can create a nginx container:

```
$ docker run --security-opt="apparmor:docker-nginx-sample" -p 80:80 --
rm -it nginx bash
```

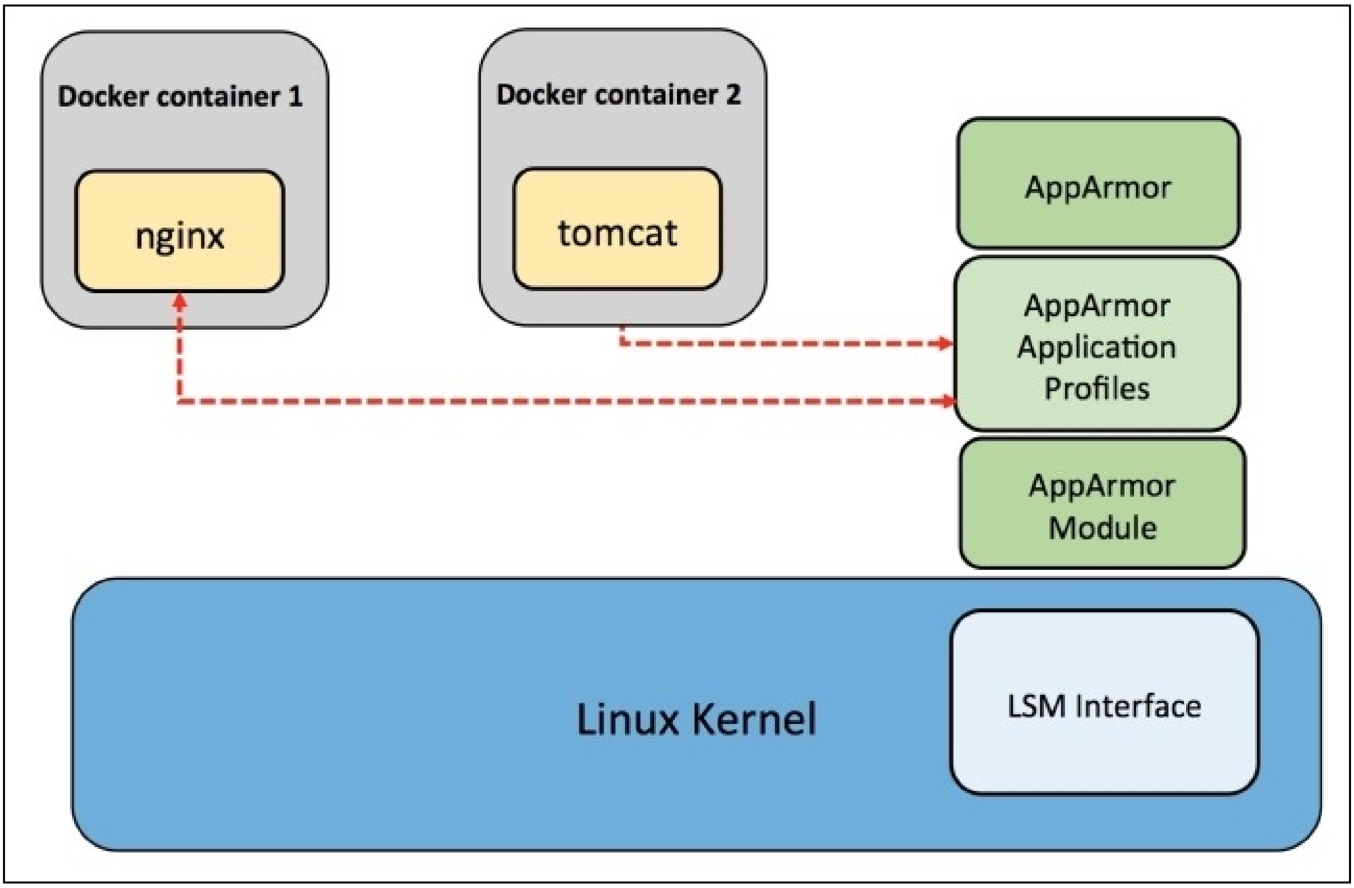
Note, if AppArmor is not able to find the file, copy the file into the `/etc/apparmor.d` directory and reload the AppArmour profiles:

```
$ sudo invoke-rc.d apparmor reload
```

Create the nginx container with the AppArmor profile:

```
ubuntu@ubuntu:~/go/src/github.com$ docker run --security-
opt="apparmor:docker-nginx-sample" -p 80:80 --rm -it nginx bash
root@84d617972e04:/# ping 8.8.8.8
ping: Lacking privilege for raw socket.
```

The following figure shows how an nginx app running inside a container uses AppArmour application profiles:


Docker container 1
nginx
Docker container 2
tomcat
AppArmor
AppArmor Application Profiles
AppArmor Module
Linux Kernel
LSM Interface

Docker security benchmark

The following tutorial shows some of the important guidelines that should be followed in order to run Docker containers in secured and production environments. It is referred from the CIS Docker Security Benchmark https://benchmarks.cisecurity.org/tools2/docker/CIS_Docker_1.6_Benchmark_v1.0.0.pdf.

Audit Docker daemon regularly

Apart from auditing your regular Linux filesystem and system calls, audit Docker daemon as well. Docker daemon runs with root privileges. It is thus necessary to audit its activities and usage:

```
$ apt-get install auditd
Reading package lists… Done
Building dependency tree
Reading state information… Done
The following extra packages will be installed:
  libauparse0
Suggested packages:
  audispd-plugins
The following NEW packages will be installed:
  auditd libauparse0
0 upgraded, 2 newly installed, 0 to remove and 50 not upgraded.
Processing triggers for libc-bin (2.21-0ubuntu4) ...
Processing triggers for ureadahead (0.100.0-19) ...
Processing triggers for systemd (225-1ubuntu9) ...
```

Remove the audit log file, if it exists:

```
$ cd /etc/audit/
$ ls
audit.log
$ nano audit.log
$ rm -rf audit.log
```

Add the audit rules for the Docker service and audit the Docker service:

```
$ nano audit.rules
-w /usr/bin/docker -k docker
$ service auditd restart
$ ausearch -k docker
<no matches>
$ docker ps
CONTAINER ID    IMAGE      COMMAND    CREATED    STATUS    PORTS     NAMES
$ ausearch -k docker
----
time->Fri Nov 27 02:29:50 2015
type=PROCTITLE msg=audit(1448620190.716:79): proctitle=646F636B6572007073
type=PATH msg=audit(1448620190.716:79): item=1 name="/lib64/ld-linux-x86-
64.so.2" inode=398512 dev=08:01 mode=0100755 ouid=0 ogid=0 rdev=00:00
nametype=NORMAL
type=PATH msg=audit(1448620190.716:79): item=0 name="/usr/bin/docker"
inode=941134 dev=08:01 mode=0100755 ouid=0 ogid=0 rdev=00:00
nametype=NORMAL
```

```
type=CWD msg=audit(1448620190.716:79):  cwd="/etc/audit"
type=EXECVE msg=audit(1448620190.716:79): argc=2 a0="docker" a1="ps"
type=SYSCALL msg=audit(1448620190.716:79): arch=c000003e syscall=59
success=yes exit=0 a0=ca1208 a1=c958c8 a2=c8
```

Create a user for the container

Currently, mapping the container's root user to a non-root user on the host is not supported by Docker. The support for user namespace would be provided in future releases. This creates a serious user isolation issue. It is thus highly recommended to ensure that there is a non-root user created for the container and the container is run using that user.

As we can see in the following snippet, by default, the `centos` Docker image has a `user` field as blank, which means, by default, the container will get a root user during runtime, which should be avoided:

```
$ docker inspect centos
[
  {
    "Id":
"e9fa5d3a0d0e19519e66af2dd8ad6903a7288de0e995b6eafbcb38aebf2b606d",
    "RepoTags": [
      "centos:latest"
    ],
    "RepoDigests": [],
    "Parent":
"c9853740aa059d078b868c4a91a069a0975fb2652e94cc1e237ef9b961afa572",
    "Comment": "",
    "Created": "2015-10-13T23:29:04.138328589Z",
    "Container":
"eaa200e2e187340f0707085b9b4eab5658b13fd190af68c71a60f6283578172f",
    "ContainerConfig": {
      "Hostname": "7aa5783a47d5",
      "Domainname": "",
      "User": "",
      contd
```

While building the Docker image, we can provide the `test` user, the less-privileged user, in the Dockerfile, as shown in the following snippet:

```
$ cd
$ mkdir test-container
$ cd test-container/
$ cat Dockerfile
FROM centos:latest
RUN useradd test
USER test
root@ubuntu:~/test-container# docker build -t vkohli .
Sending build context to Docker daemon 2.048 kB
Step 1 : FROM centos:latest
 ---> e9fa5d3a0d0e
Step 2 : RUN useradd test
 ---> Running in 0c726d186658
 ---> 12041ebdfd3f
Removing intermediate container 0c726d186658
```

```
Step 3 : USER test
 ---> Running in 86c5e0599c72
 ---> af4ba8a0fec5
Removing intermediate container 86c5e0599c72
Successfully built af4ba8a0fec5
$ docker images | grep vkohli
vkohli    latest     af4ba8a0fec5      9 seconds ago     172.6 MB
```

When we start the Docker container, we can see that it gets a `test` user, and the `docker inspect` command also shows the default user as `test`:

```
$ docker run -it vkohli /bin/bash
[test@2ff11ee54c5f /]$ whoami
test
[test@2ff11ee54c5f /]$ exit
$ docker inspect vkohli
[
  {
    "Id":
"af4ba8a0fec558d68b4873e2a1a6d8a5ca05797e0bfbab0772bcedced15683ea",
    "RepoTags": [
        "vkohli:latest"
    ],
    "RepoDigests": [],
    "Parent":
"12041ebdfd3f38df3397a8961f82c225bddc56588e348761d3e252eec868d129",
    "Comment": "",
    "Created": "2015-11-27T14:10:49.206969614Z",
    "Container":
"86c5e0599c72285983f3c5511fdec940f70cde171f1bfb53fab08854fe6d7b12",
    "ContainerConfig": {
      "Hostname": "7aa5783a47d5",
      "Domainname": "",
      "User": "test",
      Contd..
```

Do not mount sensitive host system directories on containers

If sensitive directories are mounted in read-write mode, it would be possible to make changes to files within those sensitive directories. The changes might bring down security implications or unwarranted changes that could put the Docker host in a compromised state.

If the `/run/systemd` sensitive directory is mounted in the container then we can actually shutdown the host from the container itself:

```
$ docker run -ti -v /run/systemd:/run/systemd centos /bin/bash
[root@1aca7fe47882 /]# systemctl status docker
docker.service - Docker Application Container Engine
  Loaded: loaded (/lib/systemd/system/docker.service; enabled)
  Active: active (running) since Sun 2015-11-29 12:22:50 UTC; 21min ago
  Docs: https://docs.docker.com
 Main PID: 758
  CGroup: /system.slice/docker.service
[root@1aca7fe47882 /]# shutdown
```

It can be audited by using the following command, which returns the list of current mapped directories and whether they are mounted in read-write mode for each container instance:

```
$ docker ps -q | xargs docker inspect --format '{{ .Id }}: Volumes={{
.Volumes }} VolumesRW={{ .VolumesRW }}'
```

Do not use privileged containers

Docker supports the addition and removal of capabilities, allowing the use of a non-default profile. This may make Docker more secure through capability removal, or less secure through the addition of capabilities. It is thus recommended to remove all capabilities except those explicitly required for your container process.

As seen in the following, when we run the container without the privileged mode, we are unable to change the kernel parameters, but when we run the container in privileged mode using the `--privileged` flag, it is possible to change the kernel parameters easily, which can cause security vulnerability:

```
$ docker run -it centos /bin/bash
[root@7e1b1fa4fb89 /]#  sysctl -w net.ipv4.ip_forward=0
sysctl: setting key "net.ipv4.ip_forward": Read-only file system
$ docker run --privileged -it centos /bin/bash
[root@930aaa93b4e4 /]#  sysctl -a | wc -l
sysctl: reading key "net.ipv6.conf.all.stable_secret"
sysctl: reading key "net.ipv6.conf.default.stable_secret"
sysctl: reading key "net.ipv6.conf.eth0.stable_secret"
sysctl: reading key "net.ipv6.conf.lo.stable_secret"
638
[root@930aaa93b4e4 /]# sysctl -w net.ipv4.ip_forward=0
net.ipv4.ip_forward = 0
```

So, while auditing, it should be made sure that all the containers should not have the privileged mode set to `true`:

```
$ docker ps -q | xargs docker inspect --format '{{ .Id }}: Privileged={{
.HostConfig.Privileged }}'
930aaa93b4e44c0f647b53b3e934ce162fbd9ef1fd4ec82b826f55357f6fdf3a:
Privileged=true
```

Summary

In this chapter, we took a deep dive into Docker security with an overview of cgroups and kernel namespace. We also went over some of the aspects of filesystems and Linux capabilities, which containers leverage in order to provide more features, such as the privileged containers, but at the cost of exposing itself more on the threat side. We also saw how containers can be deployed in a secured environment in AWS ECS (EC2 container service) using proxy containers to restrict vulnerable traffic. AppArmor also provides kernel-enhancement features in order to confine applications to a limited set of resources. Leveraging their benefits to Docker containers helps us to deploy them in a secured environment. Finally, we had a quick dive into Docker security benchmarks and some of the important recommendations that can be followed during auditing and Docker deployment in the production environment.

In the next chapter, we will learn about tuning and troubleshooting in the Docker network using various tools.

Chapter 6. Next Generation Networking Stack for Docker: libnetwork

In this chapter, we will learn about a new networking stack for Docker: libnetwork, which provides a pluggable architecture with a default implementation for single and multi-host virtual networking:

- Introduction
    - Goal
    - Design
- CNM objects
    - CNM attributes
    - CNM lifecycle
- Drivers
    - Bridge driver
    - Overlay network driver
- Using overlay network with Vagrant
- Overlay network with Docker Machine and Docker Swarm
- Creating an overlay network manually and using it for containers
- Container network interface
- Calico’s libnetwork driver

Goal

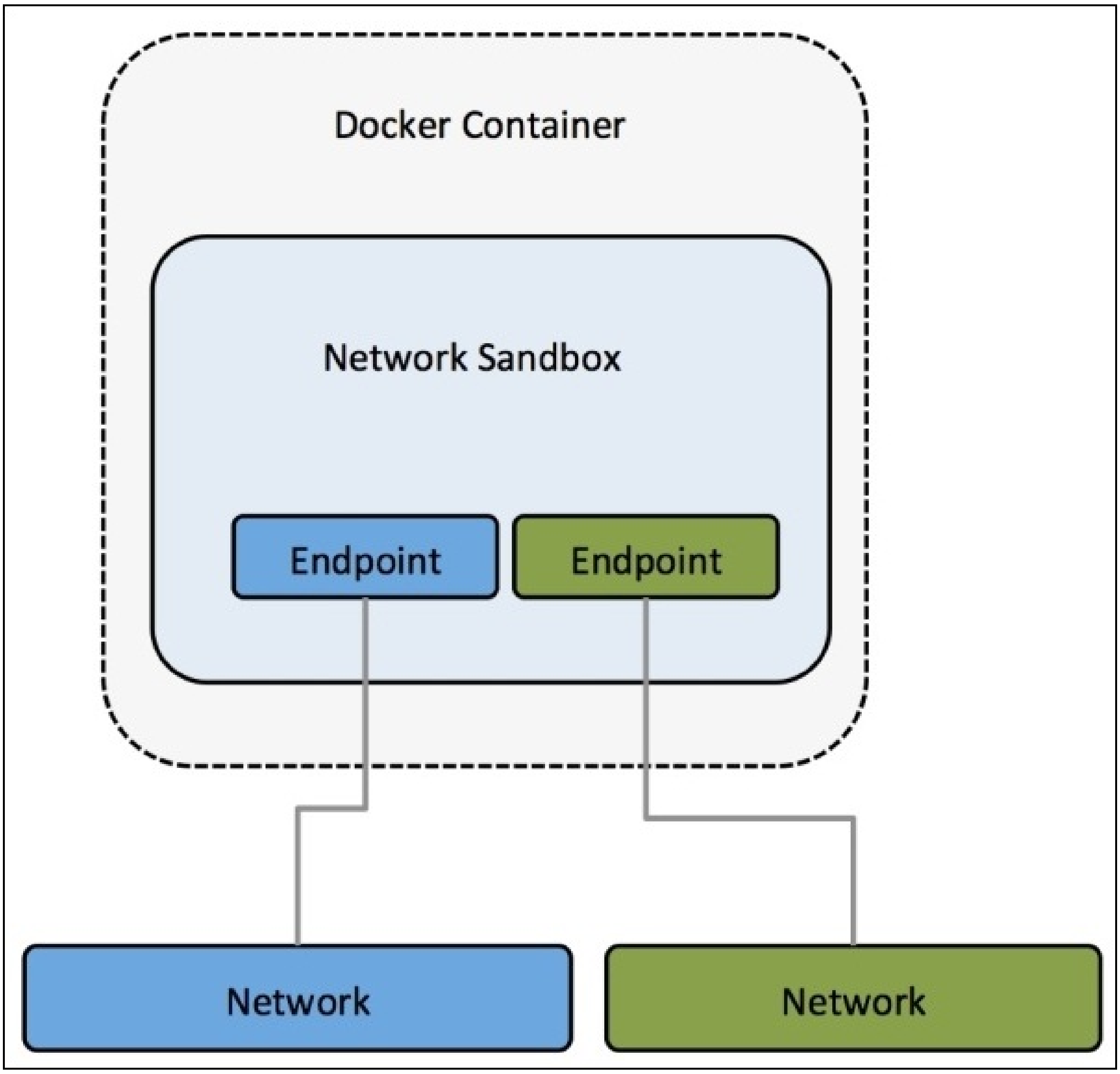
libnetwork which is written in go language is a new way for connecting Docker containers. The aim is to provide a container network model that helps programmers and provides the abstraction of network libraries. The long-term goal of libnetwork is to follow the Docker and Linux philosophy to deliver modules that work independently. libnetwork has the aim to provide a composable need for networking in containers. It also aims to modularize the networking logic in Docker Engine and libcontainer into a single, reusable library by:

- Replacing the networking module of Docker Engine with libnetwork
- Being a model that allows local and remote drivers to provide networking to containers
- Providing a tool dnet for managing and testing libnetwork—still a work in progress (reference from https://github.com/docker/libnetwork/issues/45).

Design

libnetwork implements a **container network model** (**CNM**). It formalizes the steps required to provide networking for containers, while providing an abstraction that can be used to support multiple network drivers. Its endpoint APIs are primarily used for managing the corresponding object and book-keeps them in order to provide a level of abstraction as required by the CNM model.

The CNM is built on three main components. The following figure shows the network sandbox model of libnetwork:


Docker Container
Network Sandbox
Endpoint
Endpoint
Network
Network

CNM objects

Let’s discuss the CNM objects in detail.

Sandbox

This contains the configuration of a container’s network stack, which includes management of routing tables, the container’s interface, and DNS settings. An implementation of a sandbox can be a Linux network namespace, a FreeBSD jail, or other similar concept. A sandbox may contain many endpoints from multiple networks. It also represents a container’s network configuration such as IP-address, MAC address, and DNS entries. libnetwork makes use of the OS-specific parameters to populate the network configuration represented by sandbox. libnetwork provides a framework to implement sandbox in multiple operating systems. Netlink is used to manage the routing table in namespace, and currently two implementations of sandbox exist, `namespace_linux.go` and `configure_linux.go`, to uniquely identify the path on the host filesystem.

A sandbox is associated with a single Docker container. The following data structure shows the runtime elements of a sandbox:

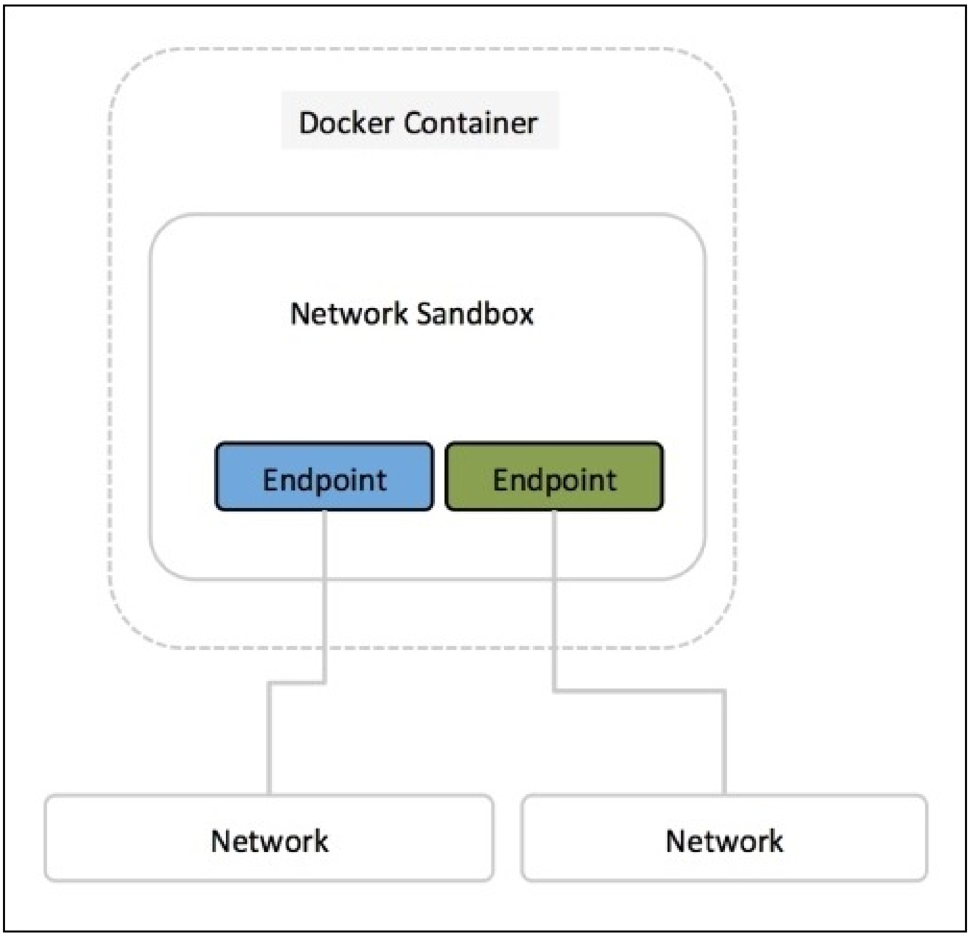
```
type sandbox struct {
  id            string
  containerID   string
  config        containerConfig
  osSbox        osl.Sandbox
  controller    *controller
  refCnt        int
  endpoints     epHeap
  epPriority    map[string]int
  joinLeaveDone chan struct{}
  dbIndex       uint64
  dbExists      bool
  isStub        bool
  inDelete      bool
  sync.Mutex
}
```

A new sandbox is instantiated from a network controller (which is explained in more detail later):

```
func (c *controller) NewSandbox(containerID string, options…SandboxOption)
(Sandbox, error) {
  …..
}
```

Endpoint

An endpoint joins a sandbox to the network and provides connectivity for services exposed by a container to the other containers deployed in the same network. It can be an internal port of Open vSwitch or a similar veth pair. An endpoint can belong to only one network but may only belong to one sandbox. An endpoint represents a service and provides various APIs to create and manage the endpoint. It has a global scope but gets attached to only one network, as shown in the following figure:



An endpoint is specified by the following data structure:

```
type endpoint struct {
  name          string
  id            string
  network       *network
  iface         *endpointInterface
  joinInfo      *endpointJoinInfo
  sandboxID     string
  exposedPorts  []types.TransportPort
  anonymous     bool
  generic       map[string]interface{}
  joinLeaveDone chan struct{}
  prefAddress   net.IP
  prefAddressV6 net.IP
  ipamOptions   map[string]string
  dbIndex       uint64
  dbExists      bool
  sync.Mutex
}
```

An endpoint is associated with a unique ID and name. It is attached to a network and a sandbox ID. It is also associated with an IPv4 and IPv6 address space. Each endpoint is associated with an `endpointInterface` struct.

Network

A network is a group of endpoints that are able to communicate with each other directly. It provides the required connectivity within the same host or multiple hosts, and whenever a network is created or updated, the corresponding driver is notified. An example is a VLAN or Linux bridge, which has a global scope within a cluster.

Networks are controlled from a network controller, which we will discuss in the next section. Every network has a name, address space, ID, and network type:

```
type network struct {
  ctrlr        *controller
  name         string
  networkType  string
  id           string
  ipamType     string
  addrSpace    string
  ipamV4Config []*IpamConf
  ipamV6Config []*IpamConf
  ipamV4Info   []*IpamInfo
  ipamV6Info   []*IpamInfo
  enableIPv6   bool
  postIPv6     bool
  epCnt        *endpointCnt
  generic      options.Generic
  dbIndex      uint64
  svcRecords   svcMap
  dbExists     bool
  persist      bool
  stopWatchCh  chan struct{}
  drvOnce      *sync.Once
  internal     bool
  sync.Mutex
}
```

Network controller

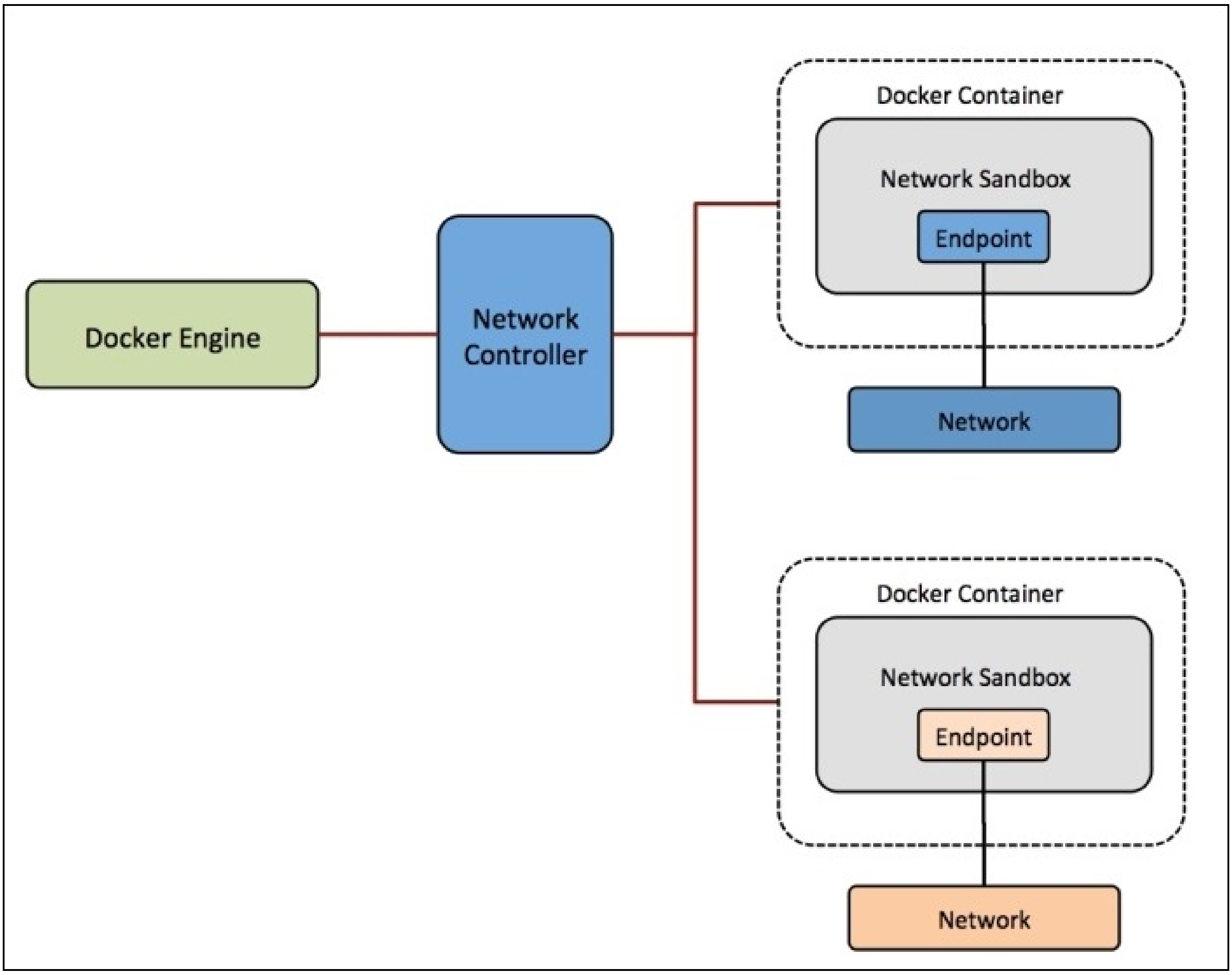
A network controller object provides APIs to create and manage a network object. It is an entry point in the libnetwork by binding a particular driver to a given network, and it supports multiple active drivers, both in-built and remote. Network controller allows users to bind a particular driver to a given network:

```
type controller struct {
  id             string
  drivers        driverTable
  ipamDrivers    ipamTable
  sandboxes      sandboxTable
  cfg            *config.Config
  stores         []datastore.DataStore
  discovery      hostdiscovery.HostDiscovery
  extKeyListener net.Listener
  watchCh        chan *endpoint
  unWatchCh      chan *endpoint
  svcDb          map[string]svcMap
  nmap           map[string]*netWatch
  defOsSbox      osl.Sandbox
  sboxOnce       sync.Once
  sync.Mutex
}
```

Each network controller has reference to the following:

- One or more drivers in the data structure driverTable
- One or more sandboxes in the data structure
- DataStore
- ipamTable

The following figure shows how **Network Controller** sits between the **Docker Engine** and the containers and networks they are attached to:


Docker Engine
Network Controller
Docker Container
Network Sandbox
Endpoint
Network
Docker Container
Network Sandbox
Endpoint
Network

CNM attributes

There are two types of attributes, as follows:

- **Options**: They are not end-user visible but are the key-value pairs of data to provide a flexible mechanism to pass driver-specific configuration from user to driver directly. libnetwork operates on the options only if the key matches a well-known label as a result value is picked up, which is represented by a generic object.
- **Labels**: They are a subset of options that are end-user variables represented in the UI using the `–labels` option. Their main function is to perform driver-specific operations and they are passed from the UI.

CNM lifecycle

Consumers of the container network model interact through the CNM objects and its APIs to network the containers that they manage.

Drivers register with network controller. Built-in drivers register inside of libnetwork, while remote drivers register with libnetwork via a plugin mechanism (WIP). Each driver handles a particular network type.

A network controller object is created using the `libnetwork.New()` API to manage the allocation of networks and optionally configure a driver with driver-specific options.

The network is created using the controller's `NewNetwork()` API by providing a name and `networkType`. The `networkType` parameter helps to choose a corresponding driver and binds the created network to that driver. From this point, any operation on the network will be handled by that driver.

The `controller.NewNetwork()` API also takes in optional options parameters that carry driver-specific options and labels, which the drivers can make use for its purpose.

`network.CreateEndpoint()` can be called to create a new endpoint in a given network. This API also accepts optional options parameters that vary with the driver.

Drivers will be called with `driver.CreateEndpoint` and it can choose to reserve IPv4/IPv6 addresses when an endpoint is created in a network. The driver will assign these addresses using the `InterfaceInfo` interface defined in the `driver` API. The IPv4/IPv6 addresses are needed to complete the endpoint as a service definition along with the ports the endpoint exposes. A service endpoint is a network address and the port number that the application container is listening on.

`endpoint.Join()` can be used to attach a container to an endpoint. The `Join` operation will create a sandbox if it doesn't exist for that container. The drivers make use of the sandbox key to identify multiple endpoints attached to the same container.

There is a separate API to create an endpoint and another to join the endpoint.

An endpoint represents a service that is independent of the container. When an endpoint is created, it has resources reserved for the container to get attached to the endpoint later. It gives a consistent networking behavior.

`endpoint.Leave()` is invoked when a container is stopped. The driver can clean up the states that it allocated during the `Join()` call. libnetwork will delete the sandbox when the last referencing endpoint leaves the network.

libnetwork keeps holding on to IP addresses as long as the endpoint is still present. These will be reused when the container (or any container) joins again. It ensures that the container's resources are re-used when they are stopped and started again.

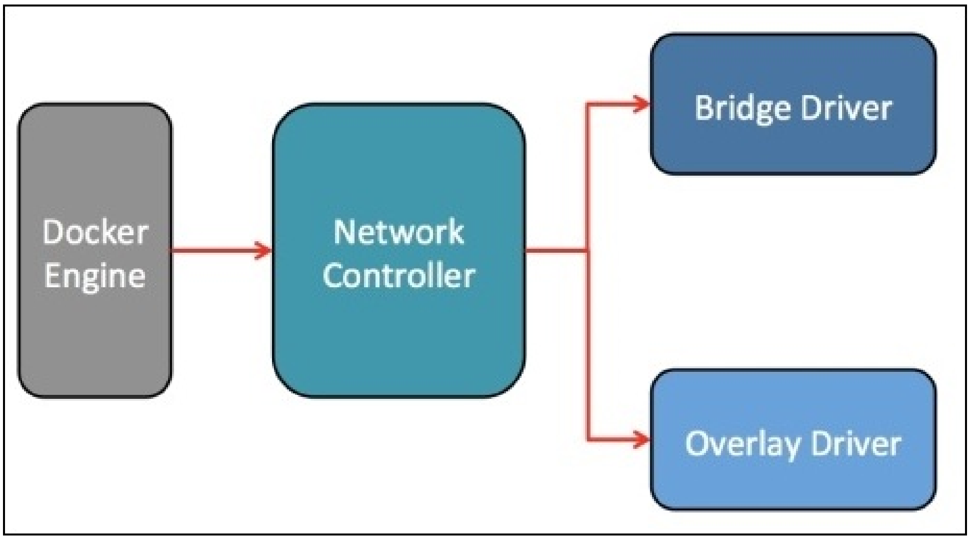
`endpoint.Delete()` is used to delete an endpoint from a network. This results in deleting the endpoint and cleaning up the cached `sandbox.Info`.

`network.Delete()` is used to delete a network. Delete is allowed if there are no endpoints attached to the network.

Driver

A driver owns a network and is responsible for making the network work and manages it. Network controller provides an API to configure the driver with specific labels/options that are not directly visible to the user but are transparent to libnetwork and can be handled by drivers directly. Drivers can be both in-built (such as bridge, host, or overlay) and remote (from plugin providers) to be deployed in various use cases and deployment scenarios.

The driver owns the network implementation and is responsible for managing it, including **IP Address Management (IPAM)**. The following figure explains the process:



The following are the in-built drivers:

- **Null**: In order to provide backward compatibility with old `docker --net=none`, this option exists primarily in the case when no networking is required.
- **Bridge**: It provides a Linux-specific bridging implementation driver.
- **Overlay**: The overlay driver implements networking that can span multiple hosts network encapsulation such as VXLAN. We will be doing a deep-dive on two of its implementations: basic setup with Consul and Vagrant setup to deploy the overlay driver.
- **Remote**: It provides a means of supporting drivers over a remote transport and a specific driver can be written as per choice.

Bridge driver

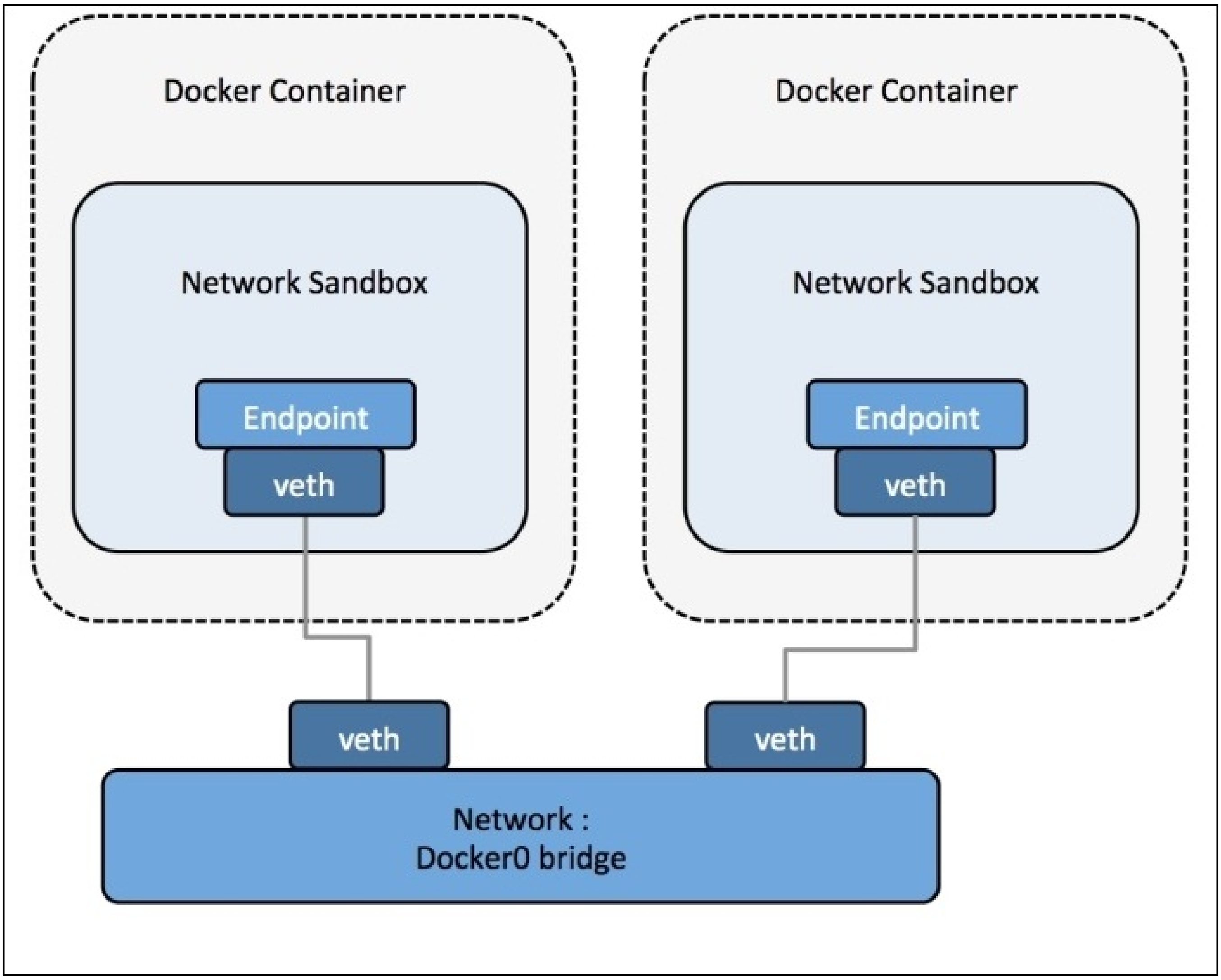
A bridge driver represents a wrapper on a Linux bridge acting as a network for libcontainer. It creates a veth pair for each network created. One end is connected to the container and the other end is connected to the bridge. The following data structure represents a bridge network:

```
type driver struct {
  config      *configuration
  etwork      *bridgeNetwork
  natChain    *iptables.ChainInfo
  filterChain *iptables.ChainInfo
  networks    map[string]*bridgeNetwork
  store       datastore.DataStore
  sync.Mutex
}
```

Some of the actions performed in a bridge driver:

- Configuring IPTables
- Managing IP forwarding
- Managing Port Mapping
- Enabling Bridge Net Filtering
- Setting up IPv4 and IPv6 on the bridge

The following diagram shows how the network is represented using `docker0` and `veth` pairs to connect endpoints with the `docker0` bridge:


Docker Container
Docker Container
Network Sandbox
Network Sandbox
Endpoint
veth
Endpoint
veth
veth
veth
Network :
Docker0 bridge

Overlay network driver

Overlay network in libnetwork uses VXLan along with a Linux bridge to create an overlaid address space. It supports multi-host networking:

```
const (
  networkType  = "overlay"
  vethPrefix   = "veth"
  vethLen      = 7
  vxlanIDStart = 256
  vxlanIDEnd   = 1000
  vxlanPort    = 4789
  vxlanVethMTU = 1450
)
type driver struct {
  eventCh      chan serf.Event
  notifyCh     chan ovNotify
  exitCh       chan chan struct{}
  bindAddress  string
  neighIP      string
  config       map[string]interface{}
  peerDb       peerNetworkMap
  serfInstance *serf.Serf
  networks     networkTable
  store        datastore.DataStore
  ipAllocator  *idm.Idm
  vxlanIdm     *idm.Idm
  once         sync.Once
  joinOnce     sync.Once
  sync.Mutex
}
```

Using overlay network with Vagrant

Overlay network is created between two containers, and VXLan tunnel connects the containers through a bridge.

Overlay network deployment Vagrant setup

This setup has been deployed using the Docker experimental version, which keeps on updating regularly and might not support some of the features:

1. Clone the official libnetwork repository and switch to the `docs` folder:

   ```
   $ git clone
   $ cd
    libnetwork/docs
   ```

2. The Vagrant script pre-exists in the repository; we will deploy the three-node setup for our Docker overlay network driver testing by using the following command:

   ```
   $ vagrant up
   Bringing machine 'consul-server' up with 'virtualbox' provider…
   Bringing machine 'net-1' up with 'virtualbox' provider…
   Bringing machine 'net-2' up with 'virtualbox' provider…
   ==> consul-server: Box 'ubuntu/trusty64' could not be found.
   Attempting to find and install…
       consul-server: Box Provider: virtualbox
       consul-server: Box Version: >= 0
   ==> consul-server: Loading metadata for box 'ubuntu/trusty64'
       consul-server: URL: https://atlas.hashicorp.com/ubuntu/trusty64
   ==> consul-server: Adding box 'ubuntu/trusty64' (v20151217.0.0) for
   provider: virtualbox
       consul-server: Downloading:
   https://atlas.hashicorp.com/ubuntu/boxes/trusty64/versions/20151217.0.0
   /providers/virtualbox.box
   ==> consul-server: Successfully added box 'ubuntu/trusty64'
   (v20151217.0.0) for 'virtualbox'!
   ==> consul-server: Importing base box 'ubuntu/trusty64'...
   ==> consul-server: Matching MAC address for NAT networking…
   ==> consul-server: Checking if box 'ubuntu/trusty64' is up to date…
   ==> consul-server: Setting the name of the VM:
   libnetwork_consul-server_1451244524836_56275
   ==> consul-server: Clearing any previously set forwarded ports…
   ==> consul-server: Clearing any previously set network interfaces…
   ==> consul-server: Preparing network interfaces based on
   configuration…
       consul-server: Adapter 1: nat
       consul-server: Adapter 2: hostonly
   ==> consul-server: Forwarding ports…
       consul-server: 22 => 2222 (adapter 1)
   ==> consul-server: Running 'pre-boot' VM customizations…
   ==> consul-server: Booting VM…
   ==> consul-server: Waiting for machine to boot. This may take a few
   minutes…
   consul-server:
   101aac79c475b84f6aff48352ead467d6b2b63ba6b64cc1b93c630489f7e3f4c
   ==> net-1: Box 'ubuntu/vivid64' could not be found. Attempting to find
   and install…
       net-1: Box Provider: virtualbox
       net-1: Box Version: >= 0
   ==> net-1: Loading metadata for box 'ubuntu/vivid64'
   ```

```
    net-1: URL: https://atlas.hashicorp.com/ubuntu/vivid64
\==> net-1: Adding box 'ubuntu/vivid64' (v20151219.0.0) for provider:
virtualbox
    net-1: Downloading:
https://atlas.hashicorp.com/ubuntu/boxes/vivid64/versions/20151219.0.0/
providers/virtualbox.box
contd…
```

3. We can list the deployed machine by Vagrant as follows:

```
$ vagrant status
Current machine states:
consul-server           running (virtualbox)
net-1                   running (virtualbox)
net-2                   running (virtualbox)
This environment represents multiple VMs. The VMs are all listed above
with their current state. For more information about a specific VM, run
`vagrant status NAME`.
```

4. The setup is complete thanks to the Vagrant script; now, we can SSH to the Docker hosts and start the testing containers:

```
$ vagrant ssh net-1
Welcome to Ubuntu 15.04 (GNU/Linux 3.19.0-42-generic x86_64)
* Documentation:https://help.ubuntu.com/
System information as of Sun Dec 27 20:04:06 UTC 2015
System load:  0.0                Users logged in:       0
Usage of /:   4.5% of 38.80GB    IP address for eth0:   10.0.2.15
Memory usage: 24%                IP address for eth1:    192.168.33.11
Swap usage:   0%                 IP address for docker0: 172.17.0.1
Processes:    78
Graph this data and manage this system at:
https://landscape.canonical.com/
Get cloud support with Ubuntu Advantage Cloud Guest:
http://www.ubuntu.com/business/services/cloud
```

5. We can create a new Docker container, and inside the container we can list the contents of the `/etc/hosts` file in order to verify that it has the overlay bridge specification, which was previously deployed, and it automatically connects to it on the launch:

```
$ docker run -it --rm ubuntu:14.04 bash
Unable to find image 'ubuntu:14.04' locally
14.04: Pulling from library/ubuntu
6edcc89ed412: Pull complete
bdf37643ee24: Pull complete
ea0211d47051: Pull complete
a3ed95caeb02: Pull complete
Digest:
sha256:d3b59c1d15c3cfb58d9f2eaab8a232f21fc670c67c11f582bc48fb32df17f3b3
Status: Downloaded newer image for ubuntu:14.04

root@65db9144c65b:/# cat /etc/hosts
172.21.0.4  2ac726b4ce60
127.0.0.1   localhost
::1 localhost ip6-localhost ip6-loopback
```

```
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
172.21.0.3  distracted_bohr
172.21.0.3  distracted_bohr.multihost
172.21.0.4  modest_curie
172.21.0.4  modest_curie.multihost
```

6. Similarly, we can create the Docker container in the other host `net-2` as well and can verify the working of the overlay network driver as both the containers will be able to ping each other in spite of being deployed on different hosts.

In the previous example, we started the Docker container with the default options and they got automatically added to a multi-host network of type overlay.

We can also creat a separate overlay bridge and add containers to it manually using the `--publish-service` option, which is part of Docker experimental:

```
vagrant@net-1:~$ docker network create -d overlay tester
447e75fd19b236e72361c270b0af4402c80e1f170938fb22183758c444966427
vagrant@net-1:~$ docker network ls
NETWORK ID          NAME                DRIVE
447e75fd19b2        tester              overlay
b77a7d741b45        bridge              bridge
40fe7cfeee20        none                null
62072090b6ac        host                host
```

The second host will also see this network and we can create containers added to the overlay network in both of these hosts by using the following option in the Docker command:

```
$ docker run -it --rm --publish-service=bar.tester.overlay ubuntu:14.04
bash
```

We will be able to verify the working of the overlay driver as both the containers will be able to ping each other. Also, tools such as tcpdump, wireshark, smartsniff, and so on can be used to capture the vXLAN package.

Overlay network with Docker Machine and Docker Swarm

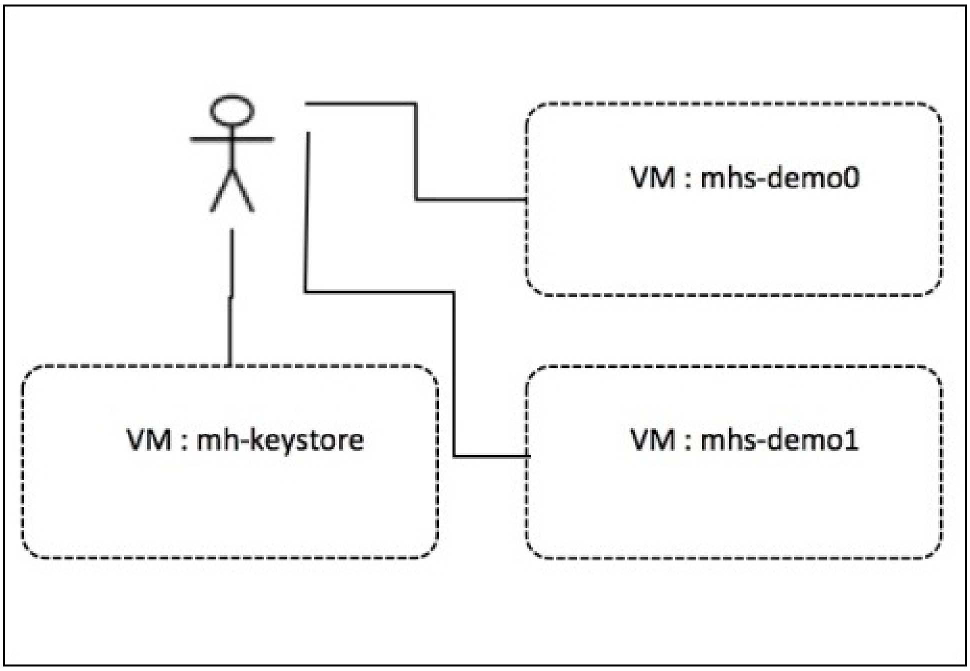
This section explains the basics of creating a multi-host network. The Docker Engine supports multi-host networking through the overlay network driver. Overlay drivers need the following pre-requisites to work:

- 3.16 Linux kernel or higher
- Access to a key-value store
- Docker supports the following key-value stores: Consul, etcd, and ZooKeeper
- A cluster of hosts connected to the key-value store
- Docker Engine daemon on each host in the cluster

This example uses Docker Machine and Docker Swarm to create the multi-network host.

Docker Machine is used to create the key-value store server and the cluster. The cluster created is a Docker Swarm cluster.

The following diagram explains how three VMs are set up using Docker Machine:



Prerequisites

- Vagrant
- Docker Engine
- Docker Machine
- Docker Swarm

Key-value store installation

An overlay network requires a key-value store. The key-value store stores information about the network state such as discovery, networks, endpoints, IP addresses, and so on. Docker supports various key-value stores such as Consul, etcd, and Zoo Keeper. This section has been implemented using Consul.

The following are the steps to install key-value store:

1. Provision a VirtualBox virtual machine called `mh-keystore`.

   When a new VM is provisioned, the process adds the Docker Engine to the host. Consul instance will be using the consul image from the Docker Hub account (https://hub.docker.com/r/progrium/consul/):

   ```
   $ docker-machine create -d virtualbox mh-keystore
   Running pre-create checks…
   Creating machine…
   (mh-keystore) Creating VirtualBox VM…
   (mh-keystore) Creating SSH key…
   (mh-keystore) Starting VM…
   Waiting for machine to be running, this may take a few minutes…
   Machine is running, waiting for SSH to be available…
   Detecting operating system of created instance…
   Detecting the provisioner…
   Provisioning with boot2docker…
   Copying certs to the local machine directory…
   Copying certs to the remote machine…
   Setting Docker configuration on the remote daemon…
   Checking connection to Docker…
   Docker is up and running!
   To see how to connect Docker to this machine, run: docker-machine env
   mh-keystore
   ```

2. Start the `progrium/consul` container created previously running on the `mh-keystore` virtual machine:

   ```
   $ docker $(docker-machine config mh-keystore) run -d \
   >     -p "8500:8500" \
   >     -h "consul" \
   >     progrium/consul -server –bootstrap

   Unable to find image 'progrium/consul:latest' locally
   latest: Pulling from progrium/consul
   3b4d28ce80e4: Pull complete
   …
   d9125e9e799b: Pull complete
   Digest:
   sha256:8cc8023462905929df9a79ff67ee435a36848ce7a10f18d6d0faba9306b97274
   Status: Downloaded newer image for progrium/consul:latest
   032884c7834ce22707ed08068c24c503d599499f1a0a58098c31be9cc84d8e6c
   ```

   A bash expansion `$(docker-machine config mh-keystore)` is used to pass the connection configuration to the Docker `run` command. The client starts a program from the `progrium/consul` image running in the `mh-keystore` machine. The container is called `consul` (flag `–h`) and is listening on port `8500` (you can choose any other port as well).

3. Set the local environment to the `mh-keystore` virtual machine:

```
$ eval "$(docker-machine env mh-keystore)"
```

4. Execute the `docker ps` command to make sure the Consul container is up:

```
$ docker ps
CONTAINER ID      IMAGE            COMMAND               CREATED
032884c7834c   progrium/consul   "/bin/start -server -"   47 seconds
ago
   STATUS          PORTS
Up 46 seconds  53/tcp, 53/udp, 8300-8302/tcp, 8301-8302/udp, 8400/tcp,
0.0.0.0:8500->8500/tcp
NAMES
sleepy_austin
```

Create a Swarm cluster with two nodes

In this step, we will use Docker Machine to provision two hosts for your network. We will create two virtual machines in VirtualBox. One of the machines will be Swarm master, which will be created first.

As each host is created, options for the overlay network driver will be passed to the Docker Engine using Swarm using the following steps:

1. Create a Swarm master virtual machine `mhs-demo0`:

   ```
   $ docker-machine create \
   -d virtualbox \
   --swarm --swarm-master \
   --swarm-discovery="consul://$(docker-machine ip mh-keystore):8500" \
   --engine-opt="cluster-store=consul://$(docker-machine ip mh-
   keystore):8500" \
   --engine-opt="cluster-advertise=eth1:2376" \
   mhs-demo0
   ```

   At creation time, you supply the engine daemon with the `--cluster-store` option. This option tells the engine the location of the key-value store for the overlay network. The bash expansion `$(docker-machine ip mh-keystore)` resolves to the IP address of the Consul server you created in step 1 of the preceding section. The `--cluster-advertise` option advertises the machine on the network.

2. Create another virtual machine `mhs-demo1` and add it to the Docker Swarm cluster:

   ```
   $ docker-machine create -d virtualbox \
       --swarm \
       --swarm-discovery="consul://$(docker-machine ip mh-keystore):8500"
   \
       --engine-opt="cluster-store=consul://$(docker-machine ip mh-
   keystore):8500" \
       --engine-opt="cluster-advertise=eth1:2376" \
   mhs-demo1

   Running pre-create checks…
   Creating machine…
   (mhs-demo1) Creating VirtualBox VM…
   (mhs-demo1) Creating SSH key…
   (mhs-demo1) Starting VM…
   Waiting for machine to be running, this may take a few minutes…
   Machine is running, waiting for SSH to be available…
   Detecting operating system of created instance…
   Detecting the provisioner…
   Provisioning with boot2docker…
   Copying certs to the local machine directory…
   Copying certs to the remote machine…
   Setting Docker configuration on the remote daemon…
   Configuring swarm…
   Checking connection to Docker…
   Docker is up and running!
   To see how to connect Docker to this machine, run: docker-machine env
   ```

```
mhs-demo1
```

3. List virtual machines using Docker Machine to confirm that they are all up and running:

```
$ docker-machine ls

NAME          ACTIVE   DRIVER       STATE     URL
SWARM                DOCKER   ERRORS
mh-keystore   *        virtualbox   Running   tcp://192.168.99.100:2376
v1.9.1
mhs-demo0     -        virtualbox   Running   tcp://192.168.99.101:2376
mhs-demo0 (master)   v1.9.1
mhs-demo1     -        virtualbox   Running   tcp://192.168.99.102:2376
mhs-demo0            v1.9.1
```

At this point, virtual machines are running. We are ready to create a multi-host network for containers using these virtual machines.

Creating an overlay network

The following command is used to create an overlay network:

```
$ docker network create --driver overlay my-net
```

We will only need to create the network on a single host in the Swarm cluster. We used the Swarm master but this command can run on any host in the Swarm cluster:

1. Check that the overlay network is running using the following command:

   ```
   $ docker network ls

   bd85c87911491d7112739e6cf08d732eb2a2841c6ca1efcc04d0b20bbb832a33
   rdua1-ltm:overlay-tutorial rdua$ docker network ls
   NETWORK ID          NAME                DRIVER
   bd85c8791149        my-net              overlay
   fff23086faa8        mhs-demo0/bridge    bridge
   03dd288a8adb        mhs-demo0/none      null
   2a706780454f        mhs-demo0/host      host
   f6152664c40a        mhs-demo1/bridge    bridge
   ac546be9c37c        mhs-demo1/none      null
   c6a2de6ba6c9       mhs-demo1/host     host
   ```

   Since we are using the Swarm master environment, we are able to see all the networks on all the Swarm agents: the default networks on each engine and the single overlay network. In this case, there are two engines running on `mhs-demo0` and `mhs-demo1`.

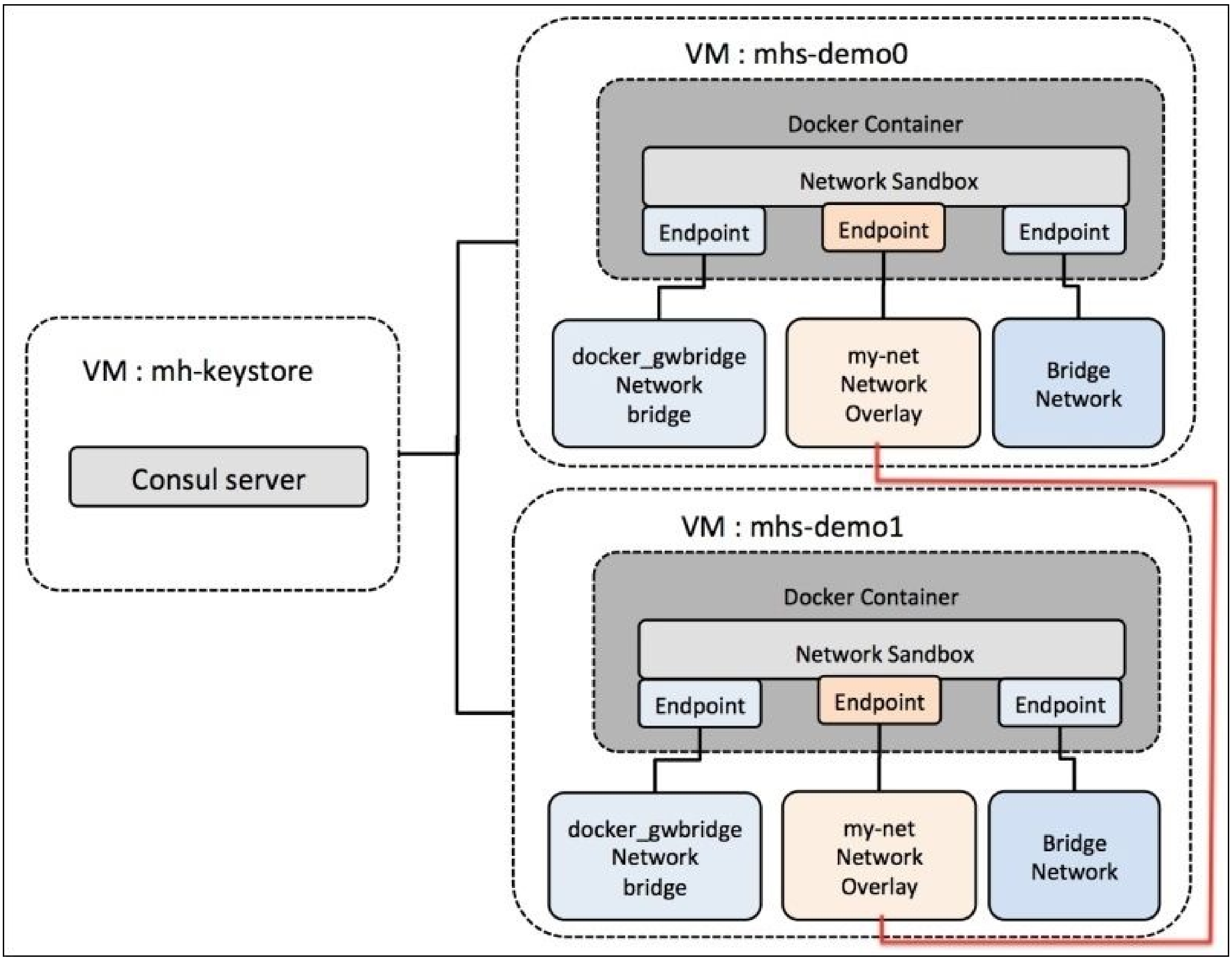
   Each `NETWORK ID` is unique.

2. Switch to each Swarm agent in turn and list the networks:

   ```
   $ eval $(docker-machine env mhs-demo0)

   $ docker network ls
   NETWORK ID          NAME                DRIVER
   bd85c8791149        my-net              overlay
   03dd288a8adb        none                  null
   2a706780454f        host                  host
   fff23086faa8        bridge              bridge

   $ eval $(docker-machine env mhs-demo1)
   $ docker network ls

   NETWORK ID          NAME                DRIVER
   bd85c8791149        my-net              overlay
   358c45b96beb        docker_gwbridge     bridge
   f6152664c40a        bridge              bridge
   ac546be9c37c        none                null
   c6a2de6ba6c9        host                host
   ```

   Both agents report they have the `my-net` network with the overlay driver. We have a multi-host overlay network running.

   The following figure shows how two containers will have containers created and tied together using the overlay `my-net`:


VM : mh-keystore
Consul server
VM : mhs-demo0
Docker Container
Network Sandbox
Endpoint
Endpoint
Endpoint
docker_gwbridge Network bridge
my-net Network Overlay
Bridge Network
VM : mhs-demo1
Docker Container
Network Sandbox
Endpoint
Endpoint
Endpoint
docker_gwbridge Network bridge
my-net Network Overlay
Bridge Network

Creating containers using an overlay network

The following are the steps for creating containers using an overlay network:

1. Create a container `c0` on `mhs-demo0` and connect to the `my-net` network:

   ```
   $ eval $(docker-machine env mhs-demo0)
   root@843b16be1ae1:/#

   $ sudo docker run -i -t --name=c0 --net=my-net  debian /bin/bash
   ```

   Execute `ifconfig` to find the IPaddress of `c0`. In this case, it is `10.0.0.4`:

   ```
   root@843b16be1ae1:/# ifconfig
   eth0      Link encap:Ethernet  HWaddr 02:42:0a:00:00:04
             inet addr:10.0.0.4  Bcast:0.0.0.0  Mask:255.255.255.0
             inet6 addr: fe80::42:aff:fe00:4/64 Scope:Link
             UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
             RX packets:17 errors:0 dropped:0 overruns:0 frame:0
             TX packets:17 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:0
             RX bytes:1474 (1.4 KB)  TX bytes:1474 (1.4 KB)

   eth1      Link encap:Ethernet  HWaddr 02:42:ac:12:00:03
             inet addr:172.18.0.3  Bcast:0.0.0.0  Mask:255.255.0.0
             inet6 addr: fe80::42:acff:fe12:3/64 Scope:Link
             UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
             RX packets:8 errors:0 dropped:0 overruns:0 frame:0
             TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:0
             RX bytes:648 (648.0 B)  TX bytes:648 (648.0 B)

   lo        Link encap:Local Loopback
             inet addr:127.0.0.1  Mask:255.0.0.0
             inet6 addr: ::1/128 Scope:Host
             UP LOOPBACK RUNNING  MTU:65536  Metric:1
             RX packets:0 errors:0 dropped:0 overruns:0 frame:0
             TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
             collisions:0 txqueuelen:0
             RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
   ```

2. Create a container, `c1` on `mhs-demo1`, and connect to the `my-net` network:

   ```
   $ eval $(docker-machine env mhs-demo1)

   $ sudo docker run -i -t --name=c1 --net=my-net  debian /bin/bash
   Unable to find image 'ubuntu:latest' locally
   latest: Pulling from library/ubuntu
   0bf056161913: Pull complete
   1796d1c62d0c: Pull complete
   e24428725dd6: Pull complete
   89d5d8e8bafb: Pull complete
   Digest:
   ```

```
sha256:a2b67b6107aa640044c25a03b9e06e2a2d48c95be6ac17fb1a387e75eebafd7c
Status: Downloaded newer image for ubuntu:latest
        root@2ce83e872408:/#
```

3. Execute `ifconfig` to find the IP address of `c1`. In this case, it is `10.0.0.3`:

```
root@2ce83e872408:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0a:00:00:03
          inet addr:10.0.0.3  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:aff:fe00:3/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:13 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:1066 (1.0 KB)  TX bytes:578 (578.0 B)

eth1      Link encap:Ethernet  HWaddr 02:42:ac:12:00:02
          inet addr:172.18.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe12:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:7 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:578 (578.0 B)  TX bytes:578 (578.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

4. Ping `c1` (`10.0.0.3`) from `c0` (`10.0.0.4`) and vice versa:

```
root@2ce83e872408:/# ping 10.0.04
PING 10.0.04 (10.0.0.4) 56(84) bytes of data.
64 bytes from 10.0.0.4: icmp_seq=1 ttl=64 time=0.370 ms
64 bytes from 10.0.0.4: icmp_seq=2 ttl=64 time=0.443 ms
64 bytes from 10.0.0.4: icmp_seq=3 ttl=64 time=0.441 ms
```

Container network interface

Container network interface (**CNI**) is a specification that defines how executable plugins can be used to configure network interfaces for Linux application containers. The official GitHub repository of CNI explains how a go library explains the implementing specification.

The container runtime first creates a new network namespace for the container in which it determines which network this container should belong to and which plugins to be executed. The network configuration is in the JSON format and defines on the container startup which plugin should be executed for the network. CNI is actually an evolving open source technology that is derived from the rkt networking protocol. Each CNI plugin is implemented as an executable and is invoked by a container management system, docker, or rkt.

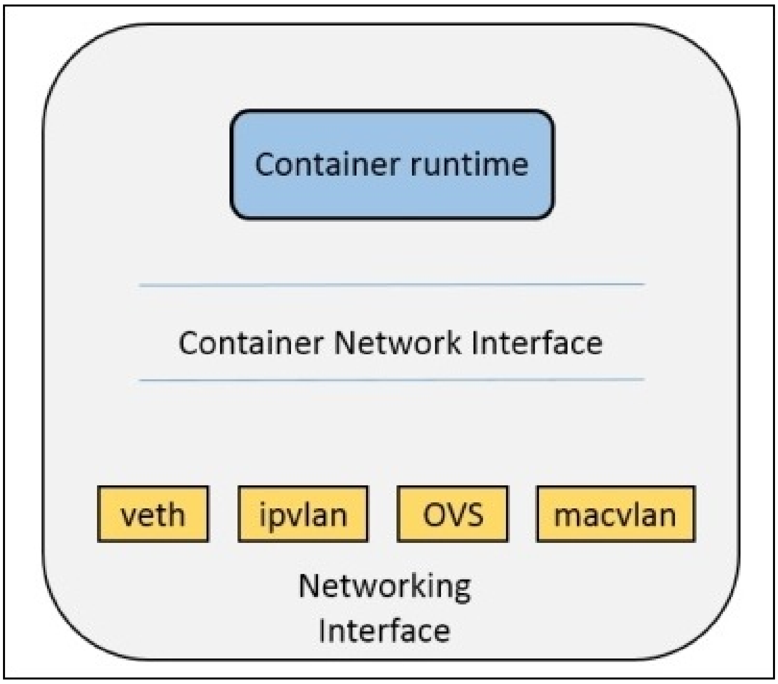
After inserting the container in the network namespace, namely by attaching one end of a veth pair to a container and attaching the other end to a bridge, it then assigns an IP to the interface and sets up routes consistent with IP address management by invoking an appropriate IPAM plugin.

The CNI model is currently used for the networking of kubelets in the Kubernetes model. Kubelets are the most important components of Kubernetes nodes, which takes the load of running containers on top of them.

The package CNI for kubelet is defined in the following Kubernetes package:

```
Constants
const (
    CNIPluginName        = "cni"
    DefaultNetDir        = "/etc/cni/net.d"
    DefaultCNIDir        = "/opt/cni/bin"
    DefaultInterfaceName = "eth0"
    VendorCNIDirTemplate = "%s/opt/%s/bin"
)
func ProbeNetworkPlugins
func ProbeNetworkPlugins(pluginDir string) []network.NetworkPlugin
```

The following figure shows the CNI placement:


Container runtime
Container Network Interface
veth
ipvlan
OVS
macvlan
Networking Interface

CNI plugin

As per the official GitHub repository (https://github.com/appc/cni), the parameters that the CNI plugin need in order to add a container to the network are:

- **Version**: The version of CNI spec that the caller is using (container call invoking the plugin).
- **Container ID**: This is optional, but recommended, and defines that there should be a unique ID across an administrative domain while the container is live. For example, the IPAM system may require that each container is allocated a unique ID so that it can be correlated properly to a container running in the background.
- **Network namespace path**: This represents the path to the network namespace to be added, for example, `/proc/[pid]/ns/net` or a `bind-mount/link` to it.
- **Network configuration**: It is the JSON document that describes a network to which a container can be joined and is explained in the following section.
- **Extra arguments**: It allows granular configuration of CNI plugins on a per-container basis.
- **Name of the interface inside the container**: It is the name that gets assigned to the container and complies with Linux restriction, which exists for interface names.

The results achieved are as follows:

- **IPs assigned to the interface**: This is either an IPv4 address or an IPv6 address assigned to the network as per requirements.
- **List of DNS nameservers**: This is a priority-ordered address list of DNS name servers.

Network configuration

The network configuration is in the JSON format that can be stored on disk or generated from other sources by container runtime. The following fields in the JSON have importance, as explained in the following:

- **cniVersion (string)**: It is Semantic Version 2.0 of the CNI specification to which this configuration meets.
- **name (string)**: It is the network name. It is unique across all containers on the host (or other administrative domain).
- **type (string)**: Refers to the filename of the CNI plugin executable.
- **ipMasq (boolean)**: Optional, sets up an IP masquerade on the host as it is necessary for the host to act as a gateway to subnets that are not able to route to the IP assigned to the container.
- **ipam**: Dictionary with IPAM-specific values.
- **type (string)**: Refers to the filename of the IPAM plugin executable.
- **routes (list)**: List of subnets (in CIDR notation) that the CNI plugin should make sure are reachable by routing through the network. Each entry is a dictionary containing:
    - **dst (string)**: A subnet in CIDR notation
    - **gw (string)**: It is the IP address of the gateway to use. If not specified, the default gateway for the subnet is assumed (as determined by the IPAM plugin).

An example configuration for plugin-specific OVS is as follows:

```
{
  "cniVersion": "0.1.0",
  "name": "pci",
  "type": "ovs",
  // type (plugin) specific
  "bridge": "ovs0",
  "vxlanID": 42,
  "ipam": {
    "type": "dhcp",
    "routes": [ { "dst": "10.3.0.0/16" }, { "dst": "10.4.0.0/16" } ]
  }
}
```

IP allocation

The CNI plugin assigns an IP address to the interface and installs necessary routes for the interface, thus it provides great flexibility for the CNI plugin and many CNI plugins internally have the same code to support several IP management schemes.

To lessen the burden on the CNI plugin, a second type of plugin, **IP address management plugin** (**IPAM**), is defined, which determines the interface IP/subnet, gateway, and routes and returns this information to the main plugin to apply. The IPAM plugin obtains information via a protocol, `ipam` section defined in the network configuration file, or data stored on the local filesystem.

IP address management interface

The IPAM plugin is invoked by running an executable, which is searched in a predefined path and is indicated by a CNI plugin via `CNI_PATH`. The IPAM plugin receives all the system environment variables from this executable, which are passed to the CNI plugin.

IPAM receives a network configuration file via stdin. Success is indicated by a zero return code and the following JSON, which gets printed to stdout (in the case of the `ADD` command):

```
{
  "cniVersion": "0.1.0",
  "ip4": {
    "ip": <ipv4-and-subnet-in-CIDR>,
    "gateway": <ipv4-of-the-gateway>,  (optional)
    "routes": <list-of-ipv4-routes>    (optional)
  },
  "ip6": {
    "ip": <ipv6-and-subnet-in-CIDR>,
    "gateway": <ipv6-of-the-gateway>,  (optional)
    "routes": <list-of-ipv6-routes>    (optional)
  },
  "dns": <list-of-DNS-nameservers>     (optional)
}
```

The following is an example of running Docker networking with CNI:

1. First, install Go Lang 1.4+ and jq (command line JSON processor) to build the CNI plugins:

   ```
   $ wget https://storage.googleapis.com/golang/go1.5.2.linux-amd64.tar.gz
   $ tar -C /usr/local -xzf go1.5.2.linux-amd64.tar.gz
   $ export PATH=$PATH:/usr/local/go/bin
   $ go version
   go version go1.5.2 linux/amd64
   $ sudo apt-get install jq
   ```

2. Clone the official CNI GitHub repository:

   ```
   $ git clone https://github.com/appc/cni.git
   Cloning into 'cni'...
   remote: Counting objects: 881, done.
   remote: Total 881 (delta 0), reused 0 (delta 0), pack-reused 881
   Receiving objects: 100% (881/881), 543.54 KiB | 313.00 KiB/s, done.
   Resolving deltas: 100% (373/373), done.
   Checking connectivity… done.
   ```

3. We will now create a `netconf` file in order to describe the network:

   ```
   mkdir -p /etc/cni/net.d
   root@rajdeepd-virtual-machine:~# cat >/etc/cni/net.d/10-mynet.conf
   <<EOF
   >{
   >  "name": "mynet",
   >  "type": "bridge",
   ```

```
>  "bridge": "cni0",
>  "isGateway": true,
>  "ipMasq": true,
>  "ipam": {
>    "type": "host-local",
>    "subnet": "10.22.0.0/16",
>    "routes": [
>      { "dst": "0.0.0.0/0" }
>    ]
>  }
>}
> EOF
```

4. Build the CNI plugins:

```
~/cni$ ./build
Building API
Building reference CLI
Building plugins
  flannel
  bridge
  ipvlan
  macvlan
  ptp
  dhcp
  host-local
```

5. Now we will execute the `priv-net-run.sh` script in order to create the private network with the CNI plugin:

```
~/cni/scripts$ sudo CNI_PATH=$CNI_PATH ./priv-net-run.sh ifconfig
eth0      Link encap:Ethernet  HWaddr 8a:72:75:7d:6d:6c
          inet addr:10.22.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::8872:75ff:fe7d:6d6c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:1 errors:0 dropped:0 overruns:0 frame:0
          TX packets:1 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:90 (90.0 B)  TX bytes:90 (90.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

6. Run a Docker container with the network namespace, which was set up previously using the CNI plugin:

```
~/cni/scripts$ sudo CNI_PATH=$CNI_PATH ./docker-run.sh --rm
busybox:latest /bin/ifconfig
eth0      Link encap:Ethernet  HWaddr 92:B2:D3:E5:BA:9B
          inet addr:10.22.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::90b2:d3ff:fee5:ba9b/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:2 errors:0 dropped:0 overruns:0 frame:0
          TX packets:2 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:180 (180.0 B)  TX bytes:168 (168.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Project Calico’s libnetwork driver

Calico provides a scalable networking solution for connecting containers, VMs, or bare metal. Calico provides connectivity using the scalable IP networking principle as a layer 3 approach. Calico can be deployed without overlays or encapsulation. The Calico service should be deployed as a container on each node and provides each container with its own IP address. It also handles all the necessary IP routing, security policy rules, and distribution of routes across a cluster of nodes.

The Calico architecture contains four important components in order to provide a better networking solution:

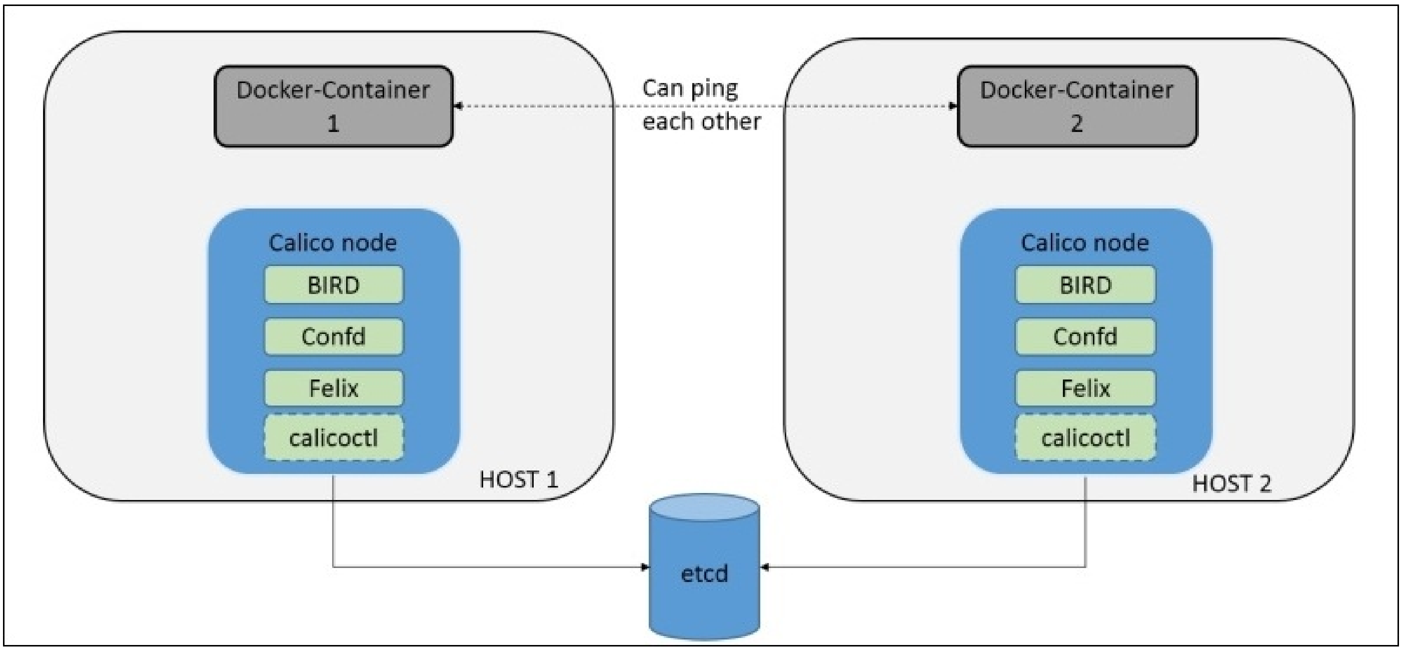
- Felix, the Calico worker process, is the heart of Calico networking, which primarily routes and provides desired connectivity to and from the workloads on host. It also provides the interface to kernels for outgoing endpoint traffic.
- BIRD, the route distribution open source BGP, exchanges routing information between hosts. The kernel endpoints, which are picked up by BIRD, are distributed to BGP peers in order to provide inter-host routing. Two BIRD processes run in the calico-node container, IPv4 (bird) and one for IPv6 (bird6).
- Confd, a templating process to auto-generate configuration for BIRD, monitors the etcd store for any changes to BGP configuration such as log levels and IPAM information. Confd also dynamically generates BIRD configuration files based on data from etcd and triggers automatically as updates are applied to data. Confd triggers BIRD to load new files whenever a configuration file is changed.
- calicoctl, the command line used to configure and start the Calico service, even allows the datastore (etcd) to define and apply security policy. The tool also provides the simple interface for general management of Calico configuration irrespective of whether Calico is running on VMs, containers, or bare metal. The following commands are supported at calicoctl:

```
$ calicoctlOverride the host:port of the ETCD server by setting the
environment variable ETCD_AUTHORITY [default: 127.0.0.1:2379]Usage:
calicoctl <command> [<args>...]
status            Print current status information
node              Configure the main calico/node container and
establish Calico networking
container         Configure containers and their addresses
profile           Configure endpoint profiles
endpoint          Configure the endpoints assigned to existing
containers
pool              Configure ip-pools
bgp               Configure global bgp
ipam              Configure IP address management
checksystem       Check for incompatibilities on the host system
diags             Save diagnostic information
version           Display the version of calicoctl
config            Configure low-level component configuration
See 'calicoctl <command> --help' to read about a specific subcommand.
```

As per the official GitHub page of the Calico repository (https://github.com/projectcalico/calico-containers), the following integration of Calico exists:

- Calico as a Docker network plugin
- Calico without Docker networking
- Calico with Kubernetes
- Calico with Mesos
- Calico with Docker Swarm

The following figure shows the Calico architecture:


Docker-Container 1
Can ping each other
Docker-Container 2
Calico node
BIRD
Confd
Felix
calicoctl
HOST 1
Calico node
BIRD
Confd
Felix
calicoctl
HOST 2
etcd

In the following tutorial we will run the manual set up of Calico on a single node machine with Docker 1.9, which finally brings libnetwork out of its experimental version to main release, and Calico can be configured directly without the need of other Docker experimental versions:

1. Get the etcd latest release and configure it on the default port 2379:

```
$ curl -L https://github.com/coreos/etcd/releases/download/v2.2.1/etcd-v2.2.1-linux-amd64.tar.gz -o etcd-v2.2.1-linux-amd64.tar.gz
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   606    0   606    0     0    445      0 --:--:--  0:00:01 --:--:--   446
100 7181k  100 7181k    0     0   441k      0  0:00:16  0:00:16 --:--:-- 1387k
$ tar xzvf etcd-v2.2.1-linux-amd64.tar.gz
etcd-v2.2.1-linux-amd64/
etcd-v2.2.1-linux-amd64/Documentation/
etcd-v2.2.1-linux-amd64/Documentation/04_to_2_snapshot_migration.md
etcd-v2.2.1-linux-amd64/Documentation/admin_guide.md
etcd-v2.2.1-linux-amd64/Documentation/api.md
```

```
contd..
etcd-v2.2.1-linux-amd64/etcd
etcd-v2.2.1-linux-amd64/etcdctl
etcd-v2.2.1-linux-amd64/README-etcdctl.md
etcd-v2.2.1-linux-amd64/README.md

$ cd etcd-v2.2.1-linux-amd64
$ ./etcd
2016-01-06 15:50:00.065733 I | etcdmain: etcd Version: 2.2.1
2016-01-06 15:50:00.065914 I | etcdmain: Git SHA: 75f8282
2016-01-06 15:50:00.065961 I | etcdmain: Go Version: go1.5.1
2016-01-06 15:50:00.066001 I | etcdmain: Go OS/Arch: linux/amd64
Contd..
2016-01-06 15:50:00.107972 I | etcdserver: starting server… [version:
2.2.1, cluster version: 2.2]
2016-01-06 15:50:00.508131 I | raft: ce2a822cea30bfca is starting a new
election at term 5
2016-01-06 15:50:00.508237 I | raft: ce2a822cea30bfca became candidate
at term 6
2016-01-06 15:50:00.508253 I | raft: ce2a822cea30bfca received vote
from ce2a822cea30bfca at term 6
2016-01-06 15:50:00.508278 I | raft: ce2a822cea30bfca became leader at
term 6
2016-01-06 15:50:00.508313 I | raft: raft.node: ce2a822cea30bfca
elected leader ce2a822cea30bfca at term 6
2016-01-06 15:50:00.509810 I | etcdserver: published {Name:default
ClientURLs:[http://localhost:2379 http://localhost:4001]} to cluster
7e27652122e8b2ae
```

2. Open the new terminal and configure the Docker daemon with the etcd key-value store by running the following commands:

```
$ service docker stop
$ docker daemon --cluster-store=etcd://0.0.0.0:2379
INFO[0000] [graphdriver] using prior storage driver "aufs"
INFO[0000] API listen on /var/run/docker.sock
INFO[0000] Firewalld running: false
INFO[0015] Default bridge (docker0) is assigned with an IP address
172.16.59.1/24. Daemon option --bip can be used to set a preferred IP
address
WARN[0015] Your kernel does not support swap memory limit.
INFO[0015] Loading containers: start.
.....INFO[0034] Skipping update of resolv.conf file with ipv6Enabled:
false because file was touched by user
INFO[0043] Loading containers: done.
INFO[0043] Daemon has completed initialization
INFO[0043] Docker daemon      commit=a34a1d5 execdriver=native-0.2
graphdriver=aufs version=1.9.1
INFO[0043] GET /v1.21/version
INFO[0043] GET /v1.21/version
INFO[0043] GET /events
INFO[0043] GET /v1.21/version
```

3. Now, in the new terminal, start the Calico container in the following way:

```
$ ./calicoctl node --libnetwork
```

```
No IP provided. Using detected IP: 10.22.0.1
Pulling Docker image calico/node:v0.10.0
Calico node is running with id:
79e75fa6d875777d31b8aead10c2712f54485c031df50667edb4d7d7cb6bb26c
Pulling Docker image calico/node-libnetwork:v0.5.2
Calico libnetwork driver is running with id:
bc7d65f6ab854b20b9b855abab4776056879f6edbcde9d744f218e556439997f
$ docker ps
CONTAINER ID        IMAGE                           COMMAND
CREATED             STATUS              PORTS               NAMES
7bb7a956af37        calico/node-libnetwork:v0.5.2   "./start.sh"
3 minutes ago       Up 3 minutes            calico-libnetwork
13a0314754d6        calico/node:v0.10.0             "/sbin/start_runit"
3 minutes ago       Up 3 minutes            calico-node
1f13020cc3a0        weaveworks/plugin:1.4.1
"/home/weave/plugin"   3 days ago          Up 3 minutes
weaveplugin
```

4. Create the Calico bridge using the `docker network` command recently introduced in the Docker CLI:

```
$docker network create –d calico net1
$ docker network ls
NETWORK ID          NAME                DRIVER
9b5f06307cf2        docker_gwbridge     bridge
1638f754fbaf        host                host
02b10aaa25d7        weave               weavemesh
65dc3cbcd2c0        bridge              bridge
f034d78cc423        net1                calico
```

5. Start the `busybox` container connected to the Calico `net1` bridge:

```
$docker run --net=net1 -itd --name=container1 busybox
1731629b6897145822f73726194b1f7441b6086ee568e973d8a88b554e838366
$ docker ps
CONTAINER ID        IMAGE                           COMMAND
CREATED             STATUS              PORTS               NAMES
1731629b6897        busybox                         "sh"
6 seconds ago       Up 5 seconds                            container1
7bb7a956af37        calico/node-libnetwork:v0.5.2   "./start.sh"
6 minutes ago       Up 6 minutes                            calico-
libnetwork
13a0314754d6        calico/node:v0.10.0             "/sbin/start_runit"
6 minutes ago       Up 6 minutes                            calico-node
1f13020cc3a0        weaveworks/plugin:1.4.1
"/home/weave/plugin"   3 days ago          Up 6 minutes
weaveplugin
$ docker attach 1731
/ #
/ # ifconfig
cali0     Link encap:Ethernet  HWaddr EE:EE:EE:EE:EE:EE
          inet addr:10.0.0.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::ecee:eeff:feee:eeee/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:29 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:5774 (5.6 KiB)  TX bytes:648 (648.0 B)

eth1      Link encap:Ethernet  HWaddr 02:42:AC:11:00:02
          inet addr:172.17.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe11:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:21 errors:0 dropped:0 overruns:0 frame:0
          TX packets:8 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:4086 (3.9 KiB)  TX bytes:648 (648.0 B)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Inside the container we can see that the container is now connected to the Calico bridge and can connect to the other containers deployed on the same bridge.

Summary

In this chapter, we looked into some of the deeper and more conceptual aspects of Docker networking, one of them being libnetworking, the future Docker network model that is already getting into shape with the release of Docker 1.9. While explaining libnetworking, we also studied the CNM model and its various objects and components with its implementation code snippets. Next, we looked into drivers of CNM, the prime one being the overlay driver, in detail, with deployment as part of the Vagrant setup. We also looked at the stand-alone integration of containers with the overlay network and as well with Docker Swarm and Docker Machine. In the next section, we explained about the CNI interface, its executable plugins, and a tutorial of configuring Docker networking with the CNI plugin.

In the last section, project Calico is explained in detail, which provides a scalable networking solution based out of libnetwork and provides integration with Docker, Kubernetes, Mesos, bare-metal, and VMs primarily.

Index

A

- Amazon EC2 container service (AWS ECS)
  - containers, securing / Securing containers in AWS ECS
  - securing / Securing containers in AWS ECS
- Amazon Machine Image (AMI) / Securing containers in AWS ECS
- AppArmor / AppArmor/SELinux
  - used, for securing Docker containers / Using AppArmor to secure Docker containers
  - URL / Using AppArmor to secure Docker containers
  - and Docker / AppArmor and Docker
- AWS
  - Kubernetes, deploying / Deploying Kubernetes on AWS
  - Mesosphere, deploying with DCOS / Deploying Mesos on AWS using DCOS
- AWS Console
  - URL / Deploying Kubernetes on AWS

B

- bridge driver / Driver, Bridge driver

C

- Calico
  - libnetwork driver / Project Calico’s libnetwork driver
  - Felix / Project Calico’s libnetwork driver
  - BIRD / Project Calico’s libnetwork driver
  - confd / Project Calico’s libnetwork driver
  - calicoctl / Project Calico’s libnetwork driver
  - URL / Project Calico’s libnetwork driver
- cgroups
  - about / Understanding Docker security II – cgroups
  - defining / Defining cgroups
  - need for / Why are cgroups required?
  - manual creation / Creating a cgroup manually
  - processes, attaching / Attaching processes to cgroups
  - URL / Docker and cgroups
  - using, with Docker / Docker and cgroups
- CNI plugin
  - URL / CNI plugin
  - about / CNI plugin
  - version / CNI plugin
  - Container ID / CNI plugin
  - network namespace path / CNI plugin
  - network configuration / CNI plugin, Network configuration
  - extra arguments / CNI plugin
  - interface name / CNI plugin
  - results achieved / CNI plugin
  - IP allocation / IP allocation
  - IP address management interface / IP address management interface
- CNM objects
  - about / CNM objects
  - sandbox / Sandbox
  - endpoint / Endpoint
  - network / Network
  - NetworkController / Network controller
  - attributes / CNM attributes
  - options attribute / CNM attributes
  - labels attribute / CNM attributes
  - lifecycle / CNM lifecycle
- components, Kubernetes
  - node / Kubernetes
  - master / Kubernetes
  - Kubectl / Kubernetes
  - Pod / Kubernetes

  - replication controller / Kubernetes
  - label / Kubernetes
- CONFIG_NET_NS option
  - URL / net namespace
- container network interface (CNI) / Container network interface
- container network model (CNM) / Design
- Container Network Model (CNM) / What’s new in Docker networking?
- containers
  - and external networks, communicating between / Communication between containers and external networks
  - creating, with overlay network / Creating containers using an overlay network
  - container network interface (CNI) / Container network interface
- CoreOS on Vagrant
  - running, URL / Networking with overlay networks – Flannel

D

- data center operating system (DCOS)
  - about / Mesosphere
  - used, for deploying Mesosphere on AWS / Deploying Mesos on AWS using DCOS
- DNS server
  - configuring / Configuring a DNS server
  - containers and external networks, communicating between / Communication between containers and external networks
  - SSH access, restricting from one container to another / Restricting SSH access from one container to another
- Docker
  - IP stack, configuring / Configuring the IP stack for Docker
  - used, for deploying web app / Deploying a web app using Docker
  - cgroups, using with / Docker and cgroups
  - and AppArmor / AppArmor and Docker
- docker0 bridge
  - about / The docker0 bridge
  - —net default mode / The —net default mode
  - —net=none mode / The —net=none mode
  - —net=container*$container2 mode / The —net=container:$container2 mode
  - —net=host mode / The —net=host mode
  - port mapping, in Docker container / Port mapping in Docker container
- Docker bridge
  - configuring / Configuring the Docker bridge
- Docker CNM model
  - about / The Docker CNM model
- Docker containers
  - linking / Linking Docker containers
  - links / Links
  - managing, with Marathon framework / Docker containers
  - securing, AppArmor used / Using AppArmor to secure Docker containers
  - security benchmark / Docker security benchmark
- Docker Hub account
  - URL / Key-value store installation
- Docker machine
  - overlay network, using with / Overlay network with Docker Machine and Docker Swarm
- Docker networking
  - about / Networking and Docker
  - Linux bridges / Linux bridges
  - Open vSwitch / Open vSwitch
  - NAT / NAT

  - IPtables / IPtables
  - AppArmor / AppArmor/SELinux
  - SELnux / AppArmor/SELinux
  - features / What’s new in Docker networking?
  - sandbox / Sandbox
  - endpoint / Endpoint
  - network / Network
  - and Kubernetes networking, differentiating between / Kubernetes networking and its differences to Docker networking
- Docker OVS
  - about / Docker OVS
  - VMs / Docker OVS
  - Hypervisor / Docker OVS
  - Physical Switch / Docker OVS
  - vNIC / Docker OVS
  - VIF (virtual interface) / Docker OVS
  - Virtual Switch / Docker OVS
- Docker security
  - kernel namespaces / Understanding Docker security I – kernel namespaces
  - cgroups / Understanding Docker security II – cgroups
- Docker Swarm
  - about / Docker Swarm
  - Spread strategy / Docker Swarm
  - Binpack strategy / Docker Swarm
  - random strategy / Docker Swarm
  - setup / Docker Swarm setup
  - networking / Docker Swarm networking
  - overlay network, using with / Overlay network with Docker Machine and Docker Swarm
- driver
  - about / Driver
  - null / Driver
  - bridge driver / Driver, Bridge driver
  - overlay network driver / Driver, Overlay network driver
  - remote / Driver
- dual stack / IPv6 support

E

- endpoint / Endpoint

F

- fields, JSON
    - cniVersion (string) / Network configuration
    - name (string) / Network configuration
    - type (string) / Network configuration
    - ipMasq (boolean) / Network configuration
    - ipam / Network configuration
    - routes (list) / Network configuration
    - routes (list), dst (string) / Network configuration
    - routes (list), gw (string) / Network configuration
- filesystem restrictions
    - about / Filesystem restrictions
    - read-only mount points / Read-only mount points
    - copy-on-write / Copy-on-write
- filters, Docker Swarm
    - constraints / Docker Swarm
    - affinity filter / Docker Swarm
    - port filter / Docker Swarm
    - dependency filter / Docker Swarm
    - health filter filter / Docker Swarm
- Flannel
    - about / Networking with overlay networks – Flannel

G

- GoLang
  - URL / Understanding Docker security I – kernel namespaces

I

- IAM console
  - URL / Deploying Kubernetes on AWS
- IP Address Management (IPAM) / Driver
- IP address management plugin (IPAM) / IP allocation
- IP stack, for Docker
  - configuring / Configuring the IP stack for Docker
  - IPv4 support / IPv4 support
  - IPv6 support / IPv6 support
- IPtables / IPtables
- IPv4 support / IPv4 support
- IPv6 support / IPv6 support

K

- kernel namespaces
  - about / Understanding Docker security I – kernel namespaces
  - PID (Process ID) namespace / Understanding Docker security I – kernel namespaces
  - network (net) namespace / Understanding Docker security I – kernel namespaces, net namespace
  - Inter Process Communication (IPC) namespace / Understanding Docker security I – kernel namespaces
  - Mount (MNT) namespace / Understanding Docker security I – kernel namespaces
  - Unix Time sharing System(UTS) namespace / Understanding Docker security I – kernel namespaces
  - Process ID (PID) namespace / pid namespace
- Kubernetes
  - about / Kubernetes
  - components / Kubernetes
  - deploying, on AWS / Deploying Kubernetes on AWS
  - networking / Kubernetes networking and its differences to Docker networking
- Kubernetes networking
  - and Docker networking, differentiating between / Kubernetes networking and its differences to Docker networking
- Kubernetes pod
  - deploying / Deploying the Kubernetes pod

L

- libnetwork / What’s new in Docker networking?
    - goal / Goal
    - URL / Goal
- libnetwork driver, Calico
    - about / Project Calico’s libnetwork driver
- Linux bridges / Linux bridges
- Linux capabilities
    - about / Linux capabilities
    - code, reference link / Linux capabilities
    - URL / Linux capabilities

M

- Management Console
  - URL / Securing containers in AWS ECS
- Mandatory Access Control (MAC) / Using AppArmor to secure Docker containers
- Marathon GUI
  - URL / Docker containers
- Mesosphere
  - about / Mesosphere
  - Docker containers, managing / Docker containers
  - web app, deploying with Docker / Deploying a web app using Docker
  - deploying, on AWS with DCOS / Deploying Mesos on AWS using DCOS
- multiple containers, over single host
  - about / Multiple containers over a single host
  - Weave, installing / Weave your containers
  - Weave, using / Weave your containers
- multiple host OVS
  - about / Multiple host OVS

N

- nameserver / Configuring a DNS server
- NAT / NAT
- network / Network
- network (net) namespace
  - about / net namespace
  - namespace management / Basic network namespace management
  - configuration / Network namespace configuration
- NetworkController / Network controller
- nsenter command line utility
  - URL / Creating a new user namespace

O

- Open vSwitch / Open vSwitch
- Open vSwitch (OVS)
  - about / Open vSwitch
  - single host OVS / Single host OVS
  - multiple host OVS / Multiple host OVS
- Open vSwitch (OVS) bridge
  - creating / Creating an OVS bridge
- overlay network
  - Vagrant, using with / Using overlay network with Vagrant
  - deployment Vagrant setup / Overlay network deployment Vagrant setup
  - using, with Docker machine / Overlay network with Docker Machine and Docker Swarm
  - using, with Docker Swarm / Overlay network with Docker Machine and Docker Swarm
  - prerequisites / Prerequisites
  - key-value store installation / Key-value store installation
  - creating / Creating an overlay network
  - used, for creating containers / Creating containers using an overlay network
- overlay network driver / Overlay network driver
- overlay networks / Overlay networks and underlay networks
  - used, for networking / Networking with overlay networks – Flannel

P

- Pipework
  - about / Introduction to Pipework
- Process ID (PID) namespace / pid namespace

R

- read-only mount points
    - sysfs filesystem / sysfs
    - sysfs filesystem, URL / sysfs
    - proc filesystem (procfs) / procfs
    - /dev/pts / /dev/pts
    - /sys//fs/cgroup / /sys/fs/cgroup

S

- sandbox / Sandbox
- security benchmark,Docker containers
  - about / Docker security benchmark
  - URL / Docker security benchmark
  - Docker daemon, auditing / Audit Docker daemon regularly
  - user, creating / Create a user for the container
  - host system directories mount, avoiding / Do not mount sensitive host system directories on containers
  - privileged containers, avoiding / Do not use privileged containers
- SELnux / AppArmor/SELinux
- single Open vSwitch (OVS)
  - about / Single host OVS
  - bridge, creating / Creating an OVS bridge
- Swarm cluster
  - creating with two nodes / Create a Swarm cluster with two nodes

U

- underlay networks / Overlay networks and underlay networks
- Unix domain socket
  - about / Unix domain socket
- user namespace
  - about / User namespace
  - creating / Creating a new user namespace

V

- Vagrant
    - using, with overlay network / Using overlay network with Vagrant

W

- web app
  - deploying, with Docker / Deploying a web app using Docker