Community Experience Distilled

Mastering CoreOS

Create production CoreOS clusters and master the art of deploying Container-based microservices

Sreenivas Makam

[PACKT] PUBLISHING open source* community experience distilled

Mastering CoreOS

Create production CoreOS clusters and master the art of deploying Container-based microservices

Sreenivas Makam

[PACKT] open source
community experience distilled
PUBLISHING

BIRMINGHAM - MUMBAI

Mastering CoreOS

First published: February 2016

Production reference: 1190216

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78528-812-8

www.packtpub.com

Credits

Author
Sreenivas Makam

Reviewers
Francisco Fernández Castaño
Neependra Khare

Commissioning Editor
Kunal Parikh

Acquisition Editor
Vinay Argekar

Content Development Editor
Rashmi Suvarna

Technical Editor
Bharat Patil

Copy Editor
Tasneem Fatehi

Project Coordinator
Judie Jose

Proofreader
Safis Editing

Indexer
Rekha Nair

Graphics
Kirk D'Penha
Abhinash Sahu

Production Coordinator
Melwyn Dsa

Cover Work
Melwyn Dsa

About the Author

Sreenivas Makam is currently working as a senior engineering manager at Cisco Systems, Bangalore. He has a masters in electrical engineering and around 18 years of experience in the networking industry. He has worked in both start-ups and big established companies. His interests include SDN, NFV, Network Automation, DevOps, and cloud technologies, and he likes to try out and follow open source projects in these areas. His blog can be found at `https://sreeninet.wordpress.com/` and his hacky code at `https://github.com/smakam`.

Sreenivas is part of the Docker bloggers forum and his blog articles have been published in Docker weekly newsletters. He has done the technical reviewing for *Mastering Ansible*, *Packt Publishing* and *Ansible Networking Report*, *O'Reilly Publisher*. He has also given presentations at Docker meetup in Bangalore. Sreenivas has one approved patent.

Acknowledgments

I have been very fortunate to always have good people around me and I would like to thank God for that. I would like to thank my parents for all the sacrifices they made while raising me and my sister and for providing the best possible support. Next, I would like to thank my wife, Lakshmi (Lucky), who has been my best friend for the last 13 years. I feel fortunate to have her as my wife as she is better qualified than me in almost everything. My daughter, Sasha, has been the spark in my life and I enjoy all the little interactions with her. I would like to thank my wife and daughter for being patient in dealing with my occasional tantrums. Lastly, I would like to thank my sister, brother-in-law, and niece for being there for me always.

I would like to thank my current company, Cisco, for allowing me to spend time on the book. I would also like to thank my Cisco managers and colleagues who have been very supportive to me in this effort.

I would like to thank my wife and my niece, Pallavi, for reviewing some of the chapters. My wife and daughter also helped out by drawing a rough draft of some of the pictures used in the book.

I would like to thank my editors and reviewers for guiding and correcting me throughout the book's preparation.

These acknowledgements would not be complete without thanking the folks at CoreOS and Docker for the amazing technology that they have developed, which allowed me to write about it. Lastly, a big thanks to the open source community for making software easily accessible to everyone.

About the Reviewers

Francisco Fernández Castaño works at GrapheneDB, one of the biggest graph database cloud providers, as a site reliability engineer. He has previously worked in a few start-ups in software tools to social networks. His interests are distributed systems, functional programming, databases, mathematics, and machine learning.

Neependra Khare is the author of *Docker Cookbook*, *Packt Publishing* (`https://www.packtpub.com/virtualization-and-cloud/docker-cookbook`), and has been co-organizing a Docker meetup group in Bangalore for more than a year now (`http://www.meetup.com/Docker-Bangalore`). He has more than 11 years of IT experience, and has earlier worked as a system administrator, support engineer, file system developer, and performance engineer. Currently, he is freelancing for Container technologies such as Docker, Kubernetes, Project Atomic, and CoreOS. You can visit his website at `http://neependra.net/` and follow him on Twitter. His Twitter handle is `@neependra.ss`.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit `www.PacktPub.com`.

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 `service@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

Free access for Packt account holders

If you have an account with Packt at `www.PacktPub.com`, you can use this to access PacktLib today and view 9 entirely free books. Simply use your login credentials for immediate access.

Table of Contents

Preface

Public cloud providers such as Amazon and Google have revolutionized cloud technologies and made it easier to consume for end users. The goal of CoreOS has been to create a secure and reliable cluster OS and make it easier to develop distributed applications using Containers. CoreOS's philosophy has been to keep the kernel to a bare minimum while keeping it secure, reliable, and updated. Everything on top of the kernel runs as Containers, except a few critical services. Containers have revolutionized application development as well as application deployment. Even though Containers existed for a long time, Docker revolutionized the Container technology by making it easier to consume and distribute Containers.

By making critical components of their technologies open source, CoreOS has created a major following in the CoreOS community as well as the general Linux and cloud communities. With Tectonic, CoreOS has combined all its open source technologies along with Kubernetes into one commercial offering. Cluster OS, Containers, and distributed application deployment are all pretty new and there is a lot of development happening in these areas. CoreOS is at a sweet spot, and this will encourage you to understand more about it.

This book will cover CoreOS internals and the technologies surrounding the deployment of Container-based microservices in a CoreOS cluster.

This book starts off with an overview of CoreOS and distributed application development and the related technologies around it. You will initially learn about installing CoreOS in different environments and using the CoreOS automatic update service. Next, critical CoreOS services to manage and interconnect services are covered along with networking and storage considerations. After this, the Container ecosystem, including Docker and Rkt, are covered along with Container orchestration. This book also covers how popular orchestration solutions such as OpenStack, the AWS Container service, and the Google Container engine integrate with CoreOS and Docker. Lastly, the book covers troubleshooting and production considerations for Containers, CoreOS, and distributed application development and deployment.

What this book covers

Chapter 1, *CoreOS Overview*, provides you with an overview of microservices, distributed application development concepts, a comparison of the Container OSes available in the market today, the basics of Containers, Docker and CoreOS.

Chapter 2, *Setting up the CoreOS Lab*, covers how to set up a CoreOS development environment in Vagrant, Amazon AWS, Google GCE, and Baremetal.

Chapter 3, *CoreOS Autoupdate*, covers the CoreOS release cycle, CoreOS automatic updates, and options to manage CoreOS updates in a cluster.

Chapter 4, *CoreOS Primary Services – Etcd, Systemd, and Fleet*, discusses the internals of CoreOS critical services—Etcd, Systemd, and Fleet. For each of the services, we will cover their installation, configuration, and application.

Chapter 5, *CoreOS Networking and Flannel Internals*, covers the basics of Container networking with a focus on how CoreOS does Container networking with Flannel. Docker networking and other related container networking technologies are also covered in this chapter.

Chapter 6, *CoreOS Storage Management*, tells us about the CoreOS base filesystem and partition table, the Container filesystem, and Container data volumes. Container data persistence using GlusterFS and Flocker are dealt with in detail.

Chapter 7, *Container Integration with CoreOS – Docker and Rkt*, focuses on the Container standards, advanced Docker topics, and the basics of Rkt Container runtime. The focus will be on how Docker and Rkt integrate with CoreOS.

Chapter 8, *Container Orchestration*, dives into the internals of Kubernetes, Docker, and Swarm, and also compares the available orchestration solutions in the market. It also covers commercial solutions such as the AWS Container service, Google Container engine, and Tectonic.

Chapter 9, *OpenStack Integration with Containers and CoreOS*, provides you with an overview of OpenStack and OpenStack integration with Containers and CoreOS. Details of OpenStack Magnum and Kuryr projects will also be covered.

Chapter 10, *CoreOS and Containers – Troubleshooting and Debugging*, covers the CoreOS Toolbox, Docker remote API, and logging. Container monitoring tools such as Sysdig and Cadvisor and Container logging tools such as Logentries will also be covered with practical examples.

Chapter 11, *CoreOS and Containers – Production Considerations*, discusses CoreOS and Container production considerations such as Service discovery, deployment patterns, CI/CD, automation, and security. It also covers the CoreOS and Docker roadmap.

What you need for this book

- A local machine: Linux, Windows, or Mac.
- Virtualization software: Vagrant and Virtualbox to run VMs in the local machine.
- Cloud accounts: Google cloud and AWS accounts to run VMs in the cloud.
- Open source software: CoreOS, Kubernetes, Docker, Weave, Calico, Flocker, GlusterFS, OpenStack, Sysdig, cadvisor, Ansible, and LogEntries. These open source software are used in specific chapters.

Who this book is for

If you are looking to deploy a CoreOS cluster or you already have a CoreOS cluster that you want to manage for better performance, security, and scale, this book is perfect for you. This book gives enough technical input for developers to deploy distributed applications using Containers and for administrators who want to manage the distributed CoreOS infrastructure. To follow the hands-on stuff, you need to have Google and AWS cloud accounts and also be able to run CoreOS VMs in your machine.

A basic understanding of public and private clouds, Containers, Docker, Linux, and CoreOS is required.

Conventions

In this book, you will 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: "The Service type is the most common type and is used to define a `service` with its dependencies."

A block of code is set as follows:

```
ExecStart=/usr/bin/docker run --name hello busybox /bin/sh -c "while
true; do echo Hello World; sleep 1; done"
ExecStop=/usr/bin/docker stop hello
[X-Fleet]
Global=true
```

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

```
core@core-01 /etc/systemd/system/multi-user.target.wants $ ls -la
lrwxrwxrwx 1 root root   34 Aug 12 13:25 hello1.service -> /etc/systemd/
system/hello1.service
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: "To do this, we need to go to each instance in the AWS console and select `Networking | change source/dest check | disable`."

Warnings or important notes appear in a box like this.

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 from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. 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.

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.

1
CoreOS Overview

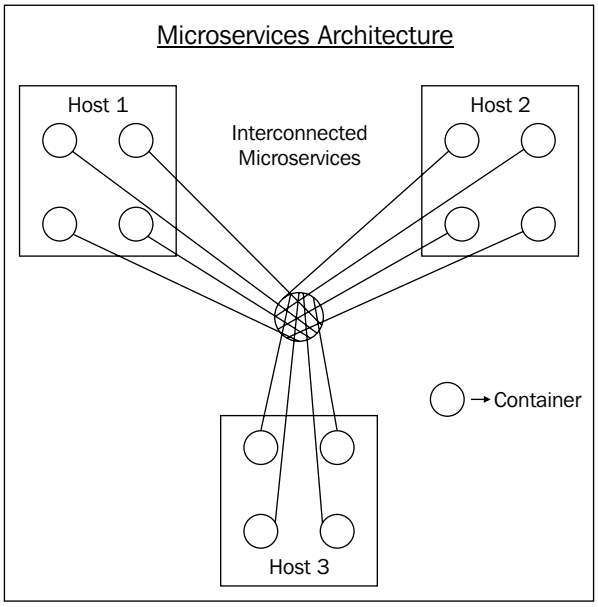
CoreOS is a Container-optimized Linux-based operating system to deploy a distributed application across a cluster of nodes. Along with providing a secure operating system, CoreOS provides services such as `etcd` and `fleet` that simplify the Container-based distributed application deployment. This chapter will provide you with an overview of Microservices and distributed application development concepts along with the basics of CoreOS, Containers, and Docker. Microservices is a software application development style where applications are composed of small, independent services talking to each other with APIs. After going through the chapter, you will be able to appreciate the role of CoreOS and Containers in the Microservices architecture.

The following topics will be covered in this chapter:

- Distributed application development—an overview and components
- Comparison of currently available minimalist Container-optimized OSes
- Containers—technology and advantages
- Docker—architecture and advantages
- CoreOS—architecture and components
- An overview of CoreOS components—`systemd`, `etcd`, `fleet`, `flannel`, and `rkt`
- Docker versus Rkt
- A workflow for distributed application development with Docker, Rkt, and CoreOS

Distributed application development

Distributed application development involves designing and coding a microservice-based application rather than creating a monolithic application. Each standalone service in the microservice-based application can be created as a Container. Distributed applications existed even before Containers were available. Containers provide the additional benefit of isolation and portability to each individual service in the distributed application. The following diagram shows you an example of a microservice-based application spanning multiple hosts:



Components of distributed application development

The following are the primary components of distributed application development. This assumes that individual services of the distributed application are created as Containers:

- Applications or microservices.
- Cloud infrastructure—public (AWS, GCE, and Digital Ocean) or private.
- Base OS—CoreOS, Atomic, Rancher OS, and others.
- Distributed data store and service discovery—`etcd`, `consul`, and `Zookeeper`.

- Load balancer—NGINX and HAProxy.
- Container runtime—Docker, Rkt, and LXC.
- Container orchestration—Fleet, Kubernetes, Mesos, and Docker Swarm.
- Storage—local or distributed storage. Some examples are GlusterFS and Ceph for cluster storage and AWS EBS for cloud storage. Flocker's upcoming storage driver plugin promises to work across different storage mechanisms.
- Networking—using cloud-based networking such as AWS VPC, CoreOS Flannel, or Docker networking.
- Miscellaneous—Container monitoring (cadvisor, Sysdig, and Newrelic) and Logging (Spout and Logentries).
- An update strategy to update microservices, such as a rolling upgrade.

Advantages and disadvantages

The following are some advantages of distributed application development:

- Application developers of each microservice can work independently. If necessary, different microservices can even have their own programming language.
- Application component reuse becomes high. Different unrelated projects can use the same microservice.
- Each individual service can be horizontally scaled. CPU and memory usage for each microservice can be tuned appropriately.
- Infrastructure can be treated like cattle rather than a pet, and it is not necessary to differentiate between each individual infrastructure component.
- Applications can be deployed in-house or on a public, private, or hybrid cloud.

The following are some problems associated with the microservices approach:

- The number of microservices to manage can become huge and this makes it complex to manage the application.
- Debugging can become difficult.
- Maintaining integrity and consistency is difficult so services must be designed to handle failures.
- Tools are constantly changing, so there is a need to stay updated with current technologies.

A minimalist Container-optimized OS

This is a new OS category for developing distributed applications that has become popular in recent years. Traditional Linux-based OSes were bulky for Container deployment and did not natively provide the services that Containers need. The following are some common characteristics of a Container-optimized OS:

- The OS needs to be bare-minimal and fast to bootup
- It should have an automated update strategy
- Application development should be done using Containers
- Redundancy and clustering should be built-in

The following table captures the comparison of features of four common Container-optimized OSes. Other OSes such as VMWare Photon and Mesos DCOS have not been included.

Feature	**CoreOS**	**Rancher OS**	**Atomic**	**Ubuntu snappy**
Company	CoreOS	Rancher Labs	Red Hat	Canonical
Containers	Docker and Rkt	Docker	Docker	Snappy packages and Docker
Maturity	First release in 2013, relatively mature	First release in early 2015, pretty new	First release in early 2015, pretty new	First release in early 2015, pretty new
Service management	Systemd and Fleet	System docker manages system services and user docker manages user containers	Systemd	Systemd and Upstart
Tools	Etcd, fleet, and flannel	Rancher has tools for service discovery, load balancing, dns, storage, and networking	Flannel and other RedHat tools	Ubuntu tools
Orchestration	Kubernetes and Tectonic	Rancher's own orchestration and Kubernetes	Kubernetes. Atomic app, and Nulecule also used	Kubernetes and any other orchestration tool
Update	Automatic, uses A and B partitions	Automatic	Automatic, uses rpm-os-tree	Automatic

Feature	CoreOS	Rancher OS	Atomic	Ubuntu snappy
Registry	Docker hub and Quay	Docker hub	Docker hub	Docker hub
Debugging	Toolbox	Rancher's own tools and external tools	RedHat tools	Ubuntu debug tools
Security	SELinux can be turned on	There is a plan to add SELinux and AppArmor support	SELinux enabled by default, additional security	AppArmor security profile can be used

Containers

Containers do virtualization at the OS level while VMs do virtualization at the hardware level. Containers in a single host share the same kernel. As Containers are lightweight, hundreds of containers can run on a single host. In a microservices-based design, the approach taken is to split a single application into multiple small independent components and run each component as a Container. LXC, Docker, and Rkt are examples of Container runtime implementations.

Technology

The following are the two critical Linux kernel technologies that are used in Containers:

- **Namespaces**: They virtualize processes, networks, filesystems, users, and so on
- **cgroups**: They limit the usage of the CPU, memory, and I/O per group of processes

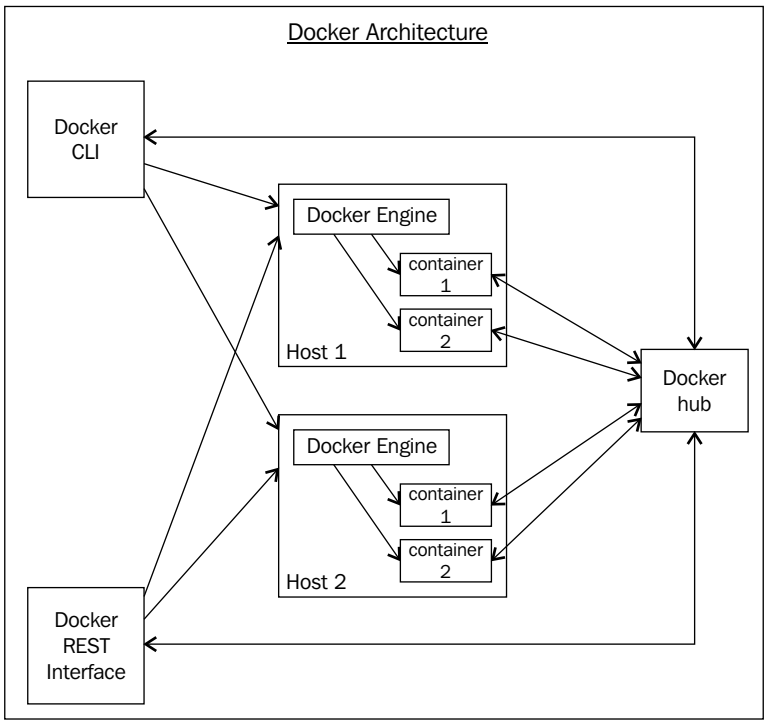
Advantages

The following are some significant advantages of Containers:

- Each container is isolated from other Containers. There is no issue of shared package management, shared libraries, and so on.
- Compared to a VM, Containers have smaller footprints and are faster to load and run.
- They provide an efficient usage of computing power.
- They can work seamlessly across dev, test, and production. This makes Containers DevOps-friendly.

An overview of Docker architecture

Docker is a Container runtime implementation. Even though Containers were available for quite a long time, Docker revolutionized Container technology by making it easier to use. The following image shows you the main components of Docker (the **Docker engine**, **Docker CLI**, **Docker REST**, and **Docker hub**) and how they interact with each other:



Following are some details on the Docker architecture:

- The Docker daemon runs in every host where Docker is installed and started.
- Docker uses Linux kernel container facilities such as namespaces and cgroups through the libcontainer library.
- The Docker client can run in the host machine or externally and it communicates with the Docker daemon using the REST interface. There is also a CLI interface that the Docker client provides.
- The Docker hub is the repository for Docker images. Both private and public images can be hosted in the Docker hub repository.

- Dockerfile is used to create container images. The following is a sample Dockerfile that is used to create a Container that starts the Apache web service exposing port `80` to the outside world:

```
FROM ubuntu:14.04
MAINTAINER Sreenivas Makam <smxxxx@yahoo.com>

RUN apt-get update

# Install apache2
RUN apt-get install -y apache2

EXPOSE 80
ENTRYPOINT ["/usr/sbin/apache2ctl"]
CMD ["-D", "FOREGROUND"]
```

- The Docker platform as of release 1.9 includes orchestration tools such as Swarm, Compose, Kitematic, and Machine as well as native networking and storage solutions. Docker follows a batteries-included pluggable approach for orchestration, storage, and networking where a native Docker solution can be swapped with vendor plugins. For example, Weave can be used as an external networking plugin, Flocker can be used as an external storage plugin, and Kubernetes can be used as an external orchestration plugin. These external plugins can replace the native Docker solutions.

Advantages of Docker

The following are some significant advantages of Docker:

- Docker has revolutionized Container packaging and tools around Containers and this has helped both application developers and infrastructure administrators
- It is easier to deploy and upgrade individual containers
- It is more suitable for the microservices architecture
- It works great across all Linux distributions as long as the kernel version is greater than or equal to 3.10
- The Union filesystem makes it faster to download and keep different versions of container images
- Container management tools such as Dockerfile, Docker engine CLI, Machine, Compose, and Swarm make it easy to manage containers
- Docker provides an easy way to share Container images using public and private registry services

CoreOS

CoreOS belongs to the minimalist Container-optimized OS category. CoreOS is the first OS in this category and many new OSes have appeared recently in the same category. CoreOS's mission is to improve the security and reliability of the Internet. CoreOS is a pioneer in this space and its first alpha release was in July 2013. A lot of developments have happened in the past two years in the area of networking, distributed storage, container runtime, authentication, and security. CoreOS is used by PaaS providers (such as Dokku and Deis), Web application development companies, and many enterprise and service providers developing distributed applications.

Properties

The following are some of the key CoreOS properties:

- The kernel is very small and fast to bootup.
- The base OS and all services are open sourced. Services can also be used standalone in non-CoreOS systems.
- No package management is provided by the OS. Libraries and packages are part of the application developed using Containers.
- It enables secure, large server clusters that can be used for distributed application development.
- It is based on principles from the Google Chrome OS.
- Container runtime, SSH, and kernel are the primary components.
- Every process is managed by systemd.
- Etcd, fleet, and flannel are all controller units running on top of the kernel.
- It supports both Docker and Rkt Container runtime.
- Automatic updates are provided with A and B partitions.
- The Quay registry service can be used to store public and private Container images.
- CoreOS release channels (stable, beta, and alpha) are used to control the release cycle.
- Commercial products include the Coreupdate service (part of the commercially managed and enterprise CoreOS), Quay enterprise, and Tectonic (CoreOS + Kubernetes).
- It currently runs on x86 processors.

Advantages

The following are some significant advantages of CoreOS:

- The kernel auto-update feature protects the kernel from security vulnerabilities.
- The CoreOS memory footprint is very small.
- The management of CoreOS machines is done at the cluster level rather than at an individual machine level.
- It provides service-level (using systemd) and node-level (using fleet) redundancy.
- Quay provides you with a private and public Container repository. The repository can be used for both Docker and Rkt containers.
- Fleet is used for basic service orchestration and Kubernetes is used for application service orchestration.
- It is supported by all major cloud providers such as AWS, GCE, Azure, and DigitalOcean.
- Majority of CoreOS components are open sourced and the customer can choose the combination of tools that is necessary for their specific application.

Supported platforms

The following are the official and community-supported CoreOS platforms. This is not an exhaustive list.

For exhaustive list of CoreOS supported platforms, please refer to this link (`https://coreos.com/os/docs/latest/`).

The platforms that are officially supported are as follows:

- Cloud platforms such as AWS, GCE, Microsoft Azure, DigitalOcean, and OpenStack
- Bare metal with PXE
- Vagrant

The platforms that are community-supported are as follows:

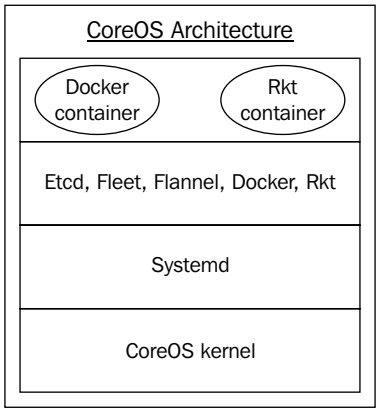
- CloudStack
- VMware

CoreOS components

The following are the CoreOS core components and CoreOS ecosystem. The ecosystem can become pretty large if automation, management, and monitoring tools are included. These have not been included here.

- Core components: Kernel, systemd, etcd, fleet, flannel, and rkt
- CoreOS ecosystem: Docker and Kubernetes

The following image shows you the different layers in the CoreOS architecture:



Kernel

CoreOS uses the latest Linux kernel in its distribution. The following screenshot shows the Linux kernel version running in the CoreOS stable release 766.3.0:

```
core@core-01 ~ $ uname -a
Linux core-01 4.1.7-coreos-r1 #2 SMP Thu Nov 5 02:10:23 UTC 2015 x86_64 Intel(R) Core(TM) i7-4800MQ CPU
@ 2.70GHz GenuineIntel GNU/Linux
```

Systemd

Systemd is an init system used by CoreOS to start, stop, and manage processes. SysVinit is one of the oldest init systems. The following are some of the common init systems used in the Unix world:

- Systemd: CoreOS and RedHat
- Upstart: Ubuntu
- Supervisord: The Python world

The following are some of the common functionality performed by an init system:

- It is the first process to start
- It controls the ordering and execution of all the user processes
- It takes care of restarting processes if they die or hang
- It takes care of process ownership and resources

The following are some specifics of systemd:

- Every process in systemd runs in one cgroup and this includes forked processes. If the systemd service is killed, all the processes associated with the service, including forked processes, are killed. This also provides you with a nice way to control resource usage. If we run a Container in systemd, we can control the resource usage even if the container contains multiple processes. Additionally, systemd takes care of restarting containers that die if we specify the `restart` option in systemd.
- Systemd units are run and controlled on a single machine.
- These are some systemd unit types—service, socket, device, and mount.
- The `Service` type is the most common type and is used to define a service with its dependencies. The `Socket` type is used to expose services to the external world. For example, `docker.service` exposes external connectivity to the Docker engine through `docker.socket`. Sockets can also be used to export logs to external machines.
- The `systemctl` CLI can be used to control Systemd units.

Systemd units

The following are some important systemd units in a CoreOS system.

Etcd2.service

The following is an example `etcd2.service` unit file:

```
[Unit]
Description=etcd2
Conflicts=etcd.service

[Service]
User=etcd
Environment=ETCD_DATA_DIR=/var/lib/etcd2
Environment=ETCD_NAME=%m
ExecStart=/usr/bin/etcd2
Restart=always
RestartSec=10s
LimitNOFILE=40000

[Install]
WantedBy=multi-user.target
```

The following are some details about the etcd2 service unit file:

- All units have the `[Unit]` and `[Install]` sections. There is a type-specific section such as `[Service]` for service units.
- The `Conflicts` option notifies that either `etcd` or `etcd2` can run, but not both.
- The `Environment` option specifies the environment variables to be used by `etcd2`. The `%m` unit specifier allows the machine ID to be taken automatically based on where the service is running.
- The `ExecStart` option specifies the executable to be run.
- The `Restart` option specifies whether the service can be restarted. The `Restartsec` option specifies the time interval after which the service should be restarted.
- `LimitNoFILE` specifies the file count limit.
- The `WantedBy` option in the `Install` section specifies the group to which this service belongs. The grouping mechanism allows systemd to start up groups of processes at the same time.

Fleet.service

The following is an example of the `fleet.service` unit file:

```
core@core-01 /usr/lib/systemd/system $ cat fleet.service
[Unit]
Description=fleet daemon

After=etcd.service
After=etcd2.service

Wants=fleet.socket
After=fleet.socket

[Service]
ExecStart=/usr/bin/fleetd
Restart=always
RestartSec=10s

[Install]
WantedBy=multi-user.target
```

In the preceding unit file, we can see two dependencies for `fleet.service. etcd. Service` and `etcd2.service` are specified as dependencies as Fleet depends on them to communicate between fleet agents in different nodes. The `fleet.socket` socket unit is also specified as a dependency as it is used by external clients to talk to Fleet.

Docker.service

The Docker service consists of the following components:

- `Docker.service`: This starts the Docker daemon
- `Docker.socket`: This allows communication with the Docker daemon from the CoreOS node
- `Docker-tcp.socket`: This allows communication with the Docker daemon from external hosts with port `2375` as the listening port

The following `docker.service` unit file starts the Docker daemon:

```
core@core-01 /usr/lib/systemd/system $ cat docker.service
[Unit]
Description=Docker Application Container Engine
Documentation=http://docs.docker.com
After=docker.socket early-docker.target network.target
Requires=docker.socket early-docker.target

[Service]
EnvironmentFile=-/run/flannel_docker_opts.env
MountFlags=slave
LimitNOFILE=1048576
LimitNPROC=1048576
ExecStart=/usr/lib/coreos/dockerd --daemon --host=fd:// $DOCKER_OPTS $DOCKER_OPT_BIP $DOCKER_OPT_MTU $DOCKER_OPT_IPMASQ

[Install]
WantedBy=multi-user.target
```

The following `docker.socket` unit file starts the local socket stream:

```
core@core-01 /usr/lib/systemd/system $ cat docker.socket
[Unit]
Description=Docker Socket for the API
PartOf=docker.service

[Socket]
ListenStream=/var/run/docker.sock
SocketMode=0660
SocketUser=docker
SocketGroup=docker

[Install]
WantedBy=sockets.target
```

Downloading the example code

You can download the example code files from your account at `http://www.packtpub.com` for all the Packt Publishing books you have purchased. If you purchased this b

The following `docker-tcp.socket` unit file sets up a listening socket for remote client communication:

```
core@core-01 /usr/lib/systemd/system $ cd -
/etc/systemd/system
core@core-01 /etc/systemd/system $ cat docker-tcp.socket
[Unit]
Description=Docker Socket for the API

[Socket]
ListenStream=2375
Service=docker.service
BindIPv6Only=both

[Install]
WantedBy=sockets.target
```

The `docker ps` command uses `docker.socket` and `docker -H tcp://127.0.0.1:2375 ps` uses `docker-tcp.socket` unit to communicate with the Docker daemon running in the local system.

The procedure to start a simple systemd service

Let's start a simple `hello1.service` unit that runs a Docker busybox container, as shown in the following image:

```
core@core-01 /etc/systemd/system $ cat hello1.service
[Unit]
Description=My Service
After=docker.service

[Service]
TimeoutStartSec=0
KillMode=none
ExecStartPre=-/usr/bin/docker kill hello1
ExecStartPre=-/usr/bin/docker rm hello1
ExecStartPre=/usr/bin/docker pull busybox
ExecStart=/usr/bin/docker run --name hello1 busybox /bin/sh -c "while true; do echo Hello World; sleep 1; done"
Restart=always
RestartSec=30s
ExecStop=/usr/bin/docker stop hello1

[Install]
WantedBy=multi-user.target
```

The following are the steps to start `hello1.service`:

1. Copy `hello1.service` as sudo to `/etc/systemd/system`.
2. Enable the service:

   `sudo systemctl enable /etc/systemd/system/hello1.service`

3. Start `hello1.service`:

   `sudo systemctl start hello1.service`

This creates the following link:

```
core@core-01 /etc/systemd/system/multi-user.target.wants $ ls -la
lrwxrwxrwx 1 root root   34 Aug 12 13:25 hello1.service -> /etc/systemd/
system/hello1.service
```

Now, we can see the status of `hello1.service`:

```
core@core-01 /etc/systemd/system $ systemctl status hello1.service
? hello1.service - My Service
   Loaded: loaded (/etc/systemd/system/hello1.service; enabled; vendor preset: disabled)
   Active: active (running) since Tue 2015-09-08 15:35:44 UTC; 3min 21s ago
  Process: 28155 ExecStop=/usr/bin/docker stop hello1 (code=exited, status=0/SUCCESS)
  Process: 28707 ExecStartPre=/usr/bin/docker pull busybox (code=exited, status=0/SUCCESS)
  Process: 28700 ExecStartPre=/usr/bin/docker rm hello1 (code=exited, status=0/SUCCESS)
  Process: 28695 ExecStartPre=/usr/bin/docker kill hello1 (code=exited, status=0/SUCCESS)
 Main PID: 28797 (docker)
   Memory: 4.4M
      CPU: 52ms
   CGroup: /system.slice/hello1.service
           └─28797 /usr/bin/docker run --name hello1 busybox /bin/sh -c while true; do echo Hello World; sleep 1; done

Sep 08 15:38:56 core-01 docker[28797]: Hello World
Sep 08 15:38:57 core-01 docker[28797]: Hello World
Sep 08 15:38:58 core-01 docker[28797]: Hello World
Sep 08 15:38:59 core-01 docker[28797]: Hello World
```

In the preceding output, we can see that the service is in the active state. At the end, we can also see `stdout` where the echo output is logged.

Let's look at the running Docker containers:

```
core@core-01 /etc/systemd/system $ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
2f52018f0de4        busybox:latest      "/bin/sh -c 'while t   4 minutes ago       Up 4 minutes                            hello1
```

When starting Docker Containers with systemd, it is necessary to avoid using the `-d` option as it prevents the Container process to be monitored by systemd. More details can be found at `https://coreos.com/os/docs/latest/getting-started-with-docker.html`.

Demonstrating systemd HA

In the `hello1.service` created, we specified two options:

```
Restart=always
RestartSec=30s
```

This means that the service should be restarted after 30 seconds in case the service exits for some reason.

Let's stop the Docker `hello1` container:

```
core@core-01 /etc/systemd/system $ docker stop hello1
hello1
core@core-01 /etc/systemd/system $ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
```

Service gets restarted automatically after 30 seconds, as shown in the following screenshot:

```
core@core-01 /etc/systemd/system $ systemctl status hello1.service
? hello1.service - My Service
   Loaded: loaded (/etc/systemd/system/hello1.service; enabled; vendor preset: disabled)
   Active: active (running) since Tue 2015-09-08 15:42:36 UTC; 29s ago
  Process: 2891 ExecStop=/usr/bin/docker stop hello1 (code=exited, status=0/SUCCESS)
  Process: 3377 ExecStartPre=/usr/bin/docker pull busybox (code=exited, status=0/SUCCESS)
  Process: 3370 ExecStartPre=/usr/bin/docker rm hello1 (code=exited, status=0/SUCCESS)
  Process: 3364 ExecStartPre=/usr/bin/docker kill hello1 (code=exited, status=0/SUCCESS)
 Main PID: 3479 (docker)
```

The following screenshot shows you that the `hello1` container is running again. From the Container status output, we can see that the container is up only for a minute:

```
core@core-01 /etc/systemd/system $ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS               NAMES
f4794367dbf7        busybox:latest      "/bin/sh -c 'while t   About a minute ago   Up About a minute                       hello1
```

We can also confirm the service restarted from the systemd logs associated with that service. In the following output, we can see that the service exited and restarted after 30 seconds:

```
Sep 08 15:42:01 core-01 systemd[1]: hello1.service: Main process exited, code=exited, status=137/n/a
Sep 08 15:42:01 core-01 docker[2891]: hello1
Sep 08 15:42:01 core-01 systemd[1]: hello1.service: Unit entered failed state.
Sep 08 15:42:01 core-01 systemd[1]: hello1.service: Failed with result 'exit-code'.
Sep 08 15:42:31 core-01 systemd[1]: hello1.service: Service hold-off time over, scheduling restart.
Sep 08 15:42:31 core-01 systemd[1]: Starting My Service...
Sep 08 15:42:31 core-01 docker[3364]: hello1
```

Etcd

Etcd is a distributed key-value store used by all machines in the CoreOS cluster to read/write and exchange data. Etcd uses the Raft consensus algorithm (`https://raft.github.io/`) to maintain a highly available cluster. Etcd is used to share configuration and monitoring data across CoreOS machines and for doing service discovery. All other CoreOS services such as Fleet and Flannel use etcd as a distributed database. Etcd can also be used as a standalone outside CoreOS. In fact, many complex distributed application projects such as Kubernetes and Cloudfoundry use etcd for their distributed key-value store. The `etcdctl` utility is the CLI frontend for etcd.

The following are two sample use cases of etcd.

- **Service discovery**: Service discovery can be used to communicate service connectivity details across containers. Let's take an example WordPress application with a WordPress application container and MySQL database container. If one of the machines has a database container and wants to communicate its service IP address and port number, it can use etcd to write the relevant key and data; the WordPress container in another host can use the key value to write to the appropriate database.
- **Configuration sharing**: The Fleet master talks to Fleet agents using etcd to decide which node in the cluster will execute the Fleet service unit.

Etcd discovery

The members in the cluster discover themselves using either a static approach or dynamic approach. In the static approach, we need to mention the IP addresses of all the neighbors statically in every node of the cluster. In the dynamic approach, we use the discovery token approach where we get a distributed token from a central etcd server and use this in all members of the cluster so that the members can discover each other.

Get a distributed token as follows:

```
curl https://discovery.etcd.io/new?size=<size>
```

The following is an example of getting a discovery token for a cluster size of three:

```
$ curl https://discovery.etcd.io/new?size=3
https://discovery.etcd.io/557c6da6ca84d7661cd21d16df3ed2d4
```

The discovery token feature is hosted by CoreOS and is implemented as an etcd cluster as well.

Cluster size

It is preferable to have an odd-sized etcd cluster as it gives a better failure tolerance. The following table shows the majority count and failure tolerance for common cluster sizes up to five. With a cluster size of two, we cannot determine majority.

Cluster size	Majority	Failure tolerance
1	1	0
3	2	1
4	3	1
5	3	2

The Majority count tells us the number of nodes that is necessary to have a working cluster, and failure tolerance tells us the number of nodes that can fail and still keep the cluster operational.

Etcd cluster details

The following screenshot shows the Etcd member list in a 3 node CoreOS cluster:

```
core@core-01 ~ $ etcdctl member list
10f62024b687f464: name=ea28e504aae84bc3968d2c66d87d6b4e peerURLs=http://172.17.8.103:2380 clientURLs=http://172.17.8.103:2379
41419684c778c117: name=18ce8345642648b9979661df12732285 peerURLs=http://172.17.8.101:2380 clientURLs=http://172.17.8.101:2379
776821cbfde4807d: name=d9ee32f6053a4fd891420ac413a7b41c peerURLs=http://172.17.8.102:2380 clientURLs=http://172.17.8.102:2379
core@core-01 ~ $
```

We can see that there are three members that are part of the etcd cluster with their machine ID, machine name, IP address, and port numbers used for etcd server-to-server and client-to-server communication.

The following output shows you the etcd cluster health:

```
core@core-01 ~ $ etcdctl cluster-health
cluster is healthy
member 10f62024b687f464 is healthy
member 41419684c778c117 is healthy
member 776821cbfde4807d is healthy
```

Here, we can see that all three members of the etcd cluster are healthy.

The following output shows you etcd statistics with the cluster leader:

```
core@core-01 ~ $ curl -L http://127.0.0.1:2379/v2/stats/self
{"name":"18ce8345642648b9979661df12732285","id":"41419684c778c117","state":"StateLeader","startTime":"2015-09-06T09:07:28.46151365Z","leaderI
nfo":{"leader":"41419684c778c117","uptime":"1h7m45.377854753s","startTime":"2015-09-06T09:08:00.004173422Z"},"recvAppendRequestCnt":0,"sendAp
pendRequestCnt":83380,"sendPkgRate":17.11027378596604,"sendBandwidthRate":1711.027378596604}core@core-01 ~ $
```

We can see that the member ID matches with the leader ID, `41419684c778c117`.

The following output shows you etcd statistics with the cluster member:

```
core@core-03 ~ $ curl -L http://127.0.0.1:2379/v2/stats/self
{"name":"ea28e504aae84bc3968d2c66d87d6b4e","id":"10f62024b687f464","state":"StateFollower","startTime":"2015-09-06T09:08:28.518269162Z","lead
erInfo":{"leader":"41419684c778c117","uptime":"1h8m8.104242437s","startTime":"2015-09-06T09:08:28.567520456Z"},"recvAppendRequestCnt":17636,"
recvPkgRate":4.120456151300817,"recvBandwidthRate":1121.9796077184562,"sendAppendRequestCnt":0}core@core-03 ~ $
```

Simple set and get operations using etcd

In the following example, we will set the `/message1` key to the `Book1` value and then later retrieve the value of the `/message1` key:

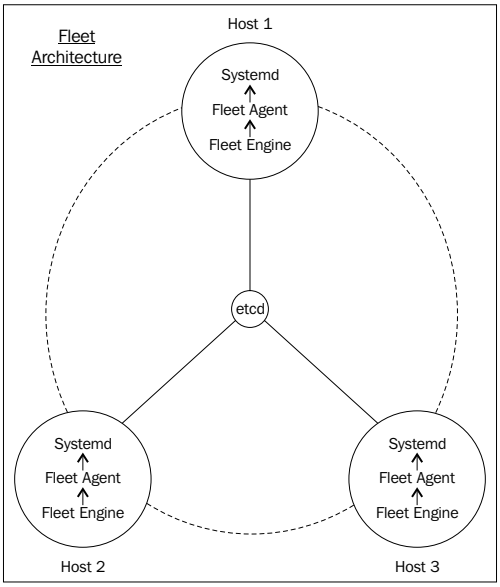
```
core@core-01 ~ $ etcdctl set /message1 Book1
etcdctl get /message1
Book1
core@core-01 ~ $ etcdctl get /message1
Book1
```

Fleet

Fleet is a cluster manager/scheduler that controls service creation at the cluster level. Like systemd being the init system for a node, Fleet serves as the init system for a cluster. Fleet uses etcd for internode communication.

The Fleet architecture

The following image shows you the components of the Fleet architecture:



- Fleet uses master, slave model with Fleet Engine playing master role and Fleet agent playing slave role. Fleet engine is responsible for scheduling Fleet units and Fleet agent is responsible for executing the units as well as reporting the status back to the Fleet engine.
- One master engine is elected among the CoreOS cluster using etcd.
- When the user starts a Fleet service, each agent bids for that service. Fleet uses a very simple `least-loaded` scheduling algorithm to schedule the unit to the appropriate node. Fleet units also consist of metadata that is useful to control where the unit runs with respect to the node property as well as based on other services running on that particular node.
- The Fleet agent processes the unit and gives it to systemd for execution.
- If any node dies, a new Fleet engine is elected and the scheduled units in that node are rescheduled to a new node. Systemd provides HA at the node level; Fleet provides HA at the cluster level.

Considering that CoreOS and Google are working closely on the Kubernetes project, a common question that comes up is the role of Fleet if Kubernetes is going to do container orchestration. Fleet is typically used for the orchestration of critical system services using systemd while Kubernetes is used for application container orchestration. Kubernetes is composed of multiple services such as the kubelet server, API server, scheduler, and replication controller and they all run as Fleet units. For smaller deployments, Fleet can also be used for application orchestration.

A Fleet scheduling example

The following is a three-node CoreOS cluster with some metadata present for each node:

```
core@core-01 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
18ce8345...     172.17.8.101    compute=web,rack=one
d9ee32f6...     172.17.8.102    compute=web,rack=two
ea28e504...     172.17.8.103    compute=db,rack=three
```

A global unit example

A global unit executes the same service unit on all the nodes in the cluster.

The following is a sample `helloglobal.service`:

```
[Unit]
Description=My Service
After=docker.service
[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill hello
ExecStartPre=-/usr/bin/docker rm hello
ExecStartPre=/usr/bin/docker pull busybox
ExecStart=/usr/bin/docker run --name hello busybox /bin/sh -c "while
true; do echo Hello World; sleep 1; done"
ExecStop=/usr/bin/docker stop hello
[X-Fleet]
Global=true
```

Let's execute the unit as follows:

```
core@core-01 ~ $ fleetctl start helloglobal.service
Triggered global unit helloglobal.service start
```

We can see that the same service is started on all three nodes:

```
core@core-01 ~ $ fleetctl list-units
UNIT                    MACHINE                         ACTIVE          SUB
helloglobal.service     18ce8345.../172.17.8.101        active          running
helloglobal.service     d9ee32f6.../172.17.8.102        activating      start-pre
helloglobal.service     ea28e504.../172.17.8.103        activating      start-pre
```

Scheduling based on metadata

Let's say that we have a three-node CoreOS cluster with the following metadata:

- Node1 (compute=web, rack=rack1)
- Node2 (compute=web, rack=rack2)
- Node3 (compute=db, rack=rack3)

We have used the `compute` metadata to identity the type of machine as `web` or `db`. We have used the `rack` metadata to identify the rack number. Fleet metadata for a node can be specified in the Fleet section of the `cloud-config`.

Let's start a web service and database service with each having its corresponding metadata and see where they get scheduled.

This is the web service:

```
[Unit]
Description=Apache web server service
After=etcd.service
After=docker.service

[Service]
TimeoutStartSec=0
KillMode=none
EnvironmentFile=/etc/environment
ExecStartPre=-/usr/bin/docker kill nginx
ExecStartPre=-/usr/bin/docker rm nginx
ExecStartPre=/usr/bin/docker pull nginx
ExecStart=/usr/bin/docker run --name nginx -p ${COREOS_PUBLIC_
IPV4}:8080:80 nginx
ExecStop=/usr/bin/docker stop nginx

[X-Fleet]
MachineMetadata=compute=web
```

This is the database service:

```
[Unit]
Description=Redis DB service
After=etcd.service
After=docker.service

[Service]
TimeoutStartSec=0
KillMode=none
EnvironmentFile=/etc/environment
ExecStartPre=-/usr/bin/docker kill redis
ExecStartPre=-/usr/bin/docker rm redis
ExecStartPre=/usr/bin/docker pull redis
ExecStart=/usr/bin/docker run --name redis redis
ExecStop=/usr/bin/docker stop redis

[X-Fleet]
MachineMetadata=compute=db
```

Let's start the services using Fleet:

```
core@core-01 ~ $ fleetctl start nginxweb.service
Unit nginxweb.service launched on 18ce8345.../172.17.8.101
core@core-01 ~ $ fleetctl start redisdb.service
Unit redisdb.service launched on ea28e504.../172.17.8.103
core@core-01 ~ $ fleetctl list-units
UNIT                    MACHINE                         ACTIVE          SUB
nginxweb.service        18ce8345.../172.17.8.101        active          running
redisdb.service         ea28e504.../172.17.8.103        activating      start-pre
```

As we can see, `nginxweb.service` got started on `Node1` and `nginxdb.service` got started on `Node3`. This is because `Node1` and `Node2` were of the `web` type and `Node3` was of the `db` type.

Fleet HA

When any of the nodes has an issue and does not respond, Fleet automatically takes care of scheduling the service units to the next appropriate machine.

From the preceding example, let's reboot `Node1`, which has `nginxweb.service`. The service gets scheduled to `Node2` and not to `Node3` because Node2 has the `web` metadata:

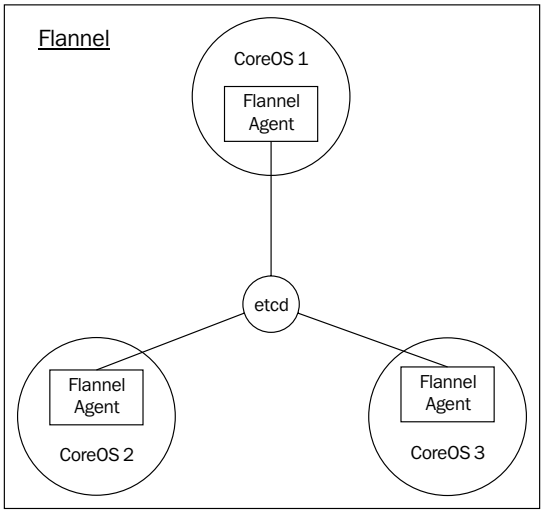
```
core@core-02 ~ $ fleetctl list-units
UNIT                    MACHINE                         ACTIVE  SUB
nginxweb.service        d9ee32f6.../172.17.8.102        active  running
redisdb.service         ea28e504.../172.17.8.103        active  running
core@core-02 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
d9ee32f6...     172.17.8.102    compute=web,rack=two
ea28e504...     172.17.8.103    compute=db,rack=three
```

In the preceding output, we can see that `nginxweb.service` is rescheduled to `Node2` and that `Node1` is not visible in the Fleet cluster.

Flannel

Flannel uses an Overlay network to allow Containers across different hosts to talk to each other. Flannel is not part of the base CoreOS image. This is done to keep the CoreOS image size minimal. When Flannel is started, the flannel container image is retrieved from the Container image repository. The Docker daemon is typically started after the Flannel service so that containers can get the IP address assigned by Flannel. This represents a chicken-and-egg problem as Docker is necessary to download the Flannel image. The CoreOS team has solved this problem by running a master Docker service whose only purpose is to download the Flannel container.

The following image shows you how Flannel agents in each node communicate using **etcd**:



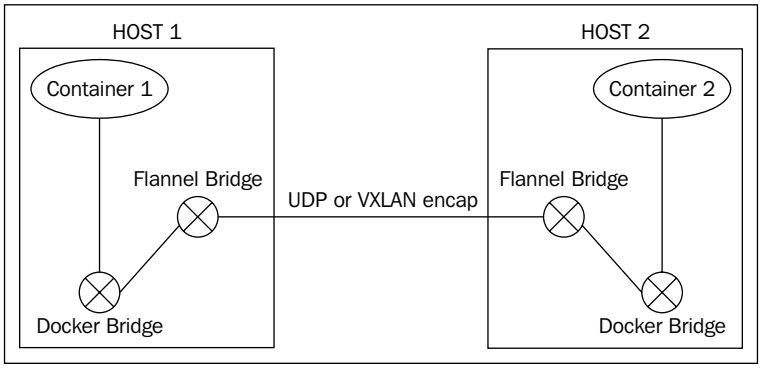
The following are some Flannel internals:

- Flannel runs without a central server and uses etcd for communication between the nodes.
- As part of starting Flannel, we need to supply a configuration file that contains the IP subnet to be used for the cluster as well as the backend protocol method (such as UDP and VXLAN). The following is a sample configuration that specifies the subnet range and backend protocol as UDP:

```
{
    "Network": "10.0.0.0/8",
    "SubnetLen": 20,
    "SubnetMin": "10.10.0.0",
    "SubnetMax": "10.99.0.0",
    "Backend": {
        "Type": "udp",
        "Port": 7890
    }
}
```

- Each node in the cluster requests an IP address range for containers created in that host and registers this IP range with etcd.
- As every node in the cluster knows the IP address range allocated for every other node, it knows how to reach containers created on any node in the cluster.
- When containers are created, containers get an IP address in the range allocated to the node.
- When Containers need to talk across hosts, Flannel does the encapsulation based on the backend encapsulation protocol chosen. Flannel, in the destination node, de-encapsulates the packet and hands it over to the Container.
- By not using port-based mapping to talk across containers, Flannel simplifies Container-to-Container communication.

The following image shows the data path for Container-to-Container communication using Flannel:



A Flannel service unit

The following is an example of a flannel service unit where we set the IP range for the flannel network as `10.1.0.0/16`:

```
- name: flanneld.service
  drop-ins:
    - name: 50-network-config.conf
      content: |
        [Service]
        ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config '{ "Network": "10.1.0.0/16" }'
  command: start
```

In a three-node etcd cluster, the following is a sample output that shows the Container IP address range picked by each node. Each node requests an IP range with a 24-bit mask. `10.1.19.0/24` is picked by node A, `10.1.3.0/24` is picked by node B, and `10.1.62.0/24` is picked by node C:

```
core@core-01 ~ $ etcdctl ls / --recursive
/coreos.com
/coreos.com/updateengine
/coreos.com/updateengine/rebootlock
/coreos.com/updateengine/rebootlock/semaphore
/coreos.com/network
/coreos.com/network/config
/coreos.com/network/subnets
/coreos.com/network/subnets/10.1.19.0-24
/coreos.com/network/subnets/10.1.3.0-24
/coreos.com/network/subnets/10.1.62.0-24
```

Rkt

Rkt is the Container runtime developed by CoreOS. Rkt does not have a daemon and is managed by systemd. Rkt uses the **Application Container image** (**ACI**) image format, which is according to the APPC specification (`https://github.com/appc/spec`). Rkt's execution is split into three stages. This approach was taken so that some of the stages can be replaced by a different implementation if needed. Following are details on the three stages of Rkt execution:

Stage 0:

This is the first stage of Container execution. This stage does image discovery, retrieval and sets up filesystem for stages 1 and 2.

Stage 1:

This stage sets up the execution environment for containers like Container namespace, cgroups using the filesystem setup by stage 0.

Stage 2:

This stage executes the Container using execution environment setup by stage 1 and filesystem setup by stage 0.

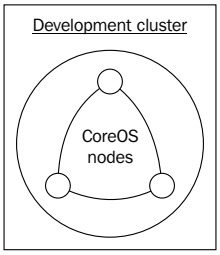
As of release 0.10.0, Rkt is still under active development and is not ready for production.

The CoreOS cluster architecture

Nodes in the CoreOS cluster are used to run critical CoreOS services such as etcd, fleet, Docker, systemd, flannel, and journald as well as application containers. It is important to avoid using the same host to run critical services as well as application containers so that there is no resource contention for critical services. This kind of scheduling can be achieved using the Fleet metadata to separate the core machines and worker machines. The following are two cluster approaches.

The development cluster

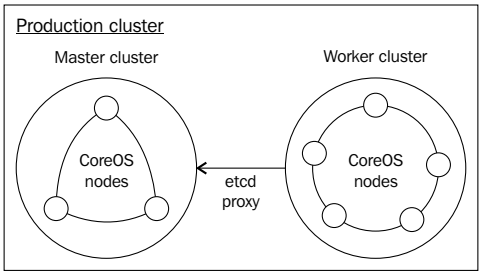
The following image shows a development cluster with three CoreOS nodes:



To try out CoreOS and etcd, we can start with a single-node cluster. With this approach, there is no need to have dynamic discovery of cluster members. Once this works fine, we can expand the cluster size to three or five to achieve redundancy. The static or dynamic discovery approach can be used to discover CoreOS members. As CoreOS critical services and application containers run in the same cluster, there could be resource contention in this approach.

The production cluster

The following image shows a production cluster with a three-node master cluster and five-node worker cluster:



We can have a three or five-node master cluster to run critical CoreOS services and then have a dynamic worker cluster to run application Containers. The master cluster will run etcd, fleet, and other critical services. In worker nodes, etcd will be set up to proxy to master nodes so that worker nodes can use master nodes for etcd communication. Fleet, in worker nodes, will also be set up to use etcd in master nodes.

Docker versus Rkt

As this is a controversial topic, I will try to give a neutral stand here.

History

CoreOS team started the Rkt project because of the following reasons:

- Container interoperability issue needed to be addressed since Docker runtime was not fully following the Container manifest specification
- Getting Docker to run under systemd had some issues because of Docker running as the daemon
- Container image discovery and image signing required improvements
- Security model for Containers needed to be improved

APPC versus OCI

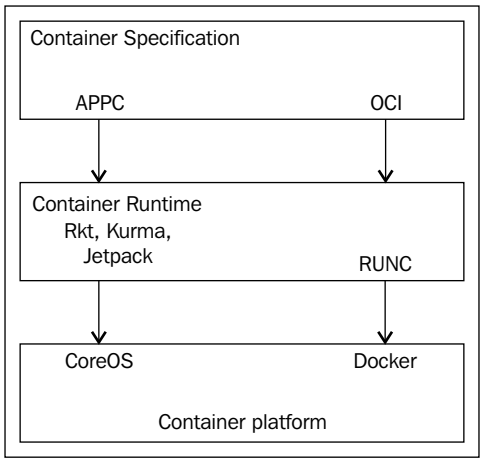
APPC (`https://github.com/appc/spec`) and OCI (`https://github.com/opencontainers/specs`) define Container standards.

The APPC specification is primarily driven by CoreOS along with a few other community members. The APPC specification defines the following:

- **Image format**: Packaging and signing
- **Runtime**: How to execute the Container
- **Naming and Sharing**: Automatic discovery

APPC is implemented by Rkt, Kurma, Jetpack, and others.

OCI (`https://www.opencontainers.org/`) is an open container initiative project started in April 2015 and has members from all major companies including Docker and CoreOS. Runc is an implementation of OCI. The following image shows you how APPC, OCI, Docker, and Rkt are related:



The current status

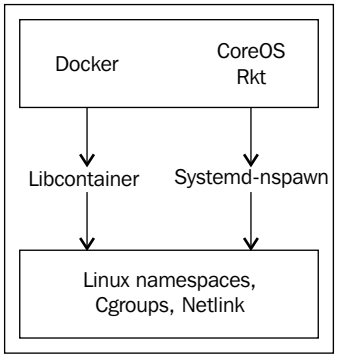
Based on the latest developments, there is consensus among the community to having a common container specification called the Open Container Specification. Anyone can develop a Container runtime based on this specification. This will allow Container images to be interoperable. Docker, Rkt, and Odin are examples of Container runtime.

The original APPC container specification proposed by CoreOS covers four different elements of container management—packaging, signing, naming (sharing the container with others), and runtime. As per the latest CoreOS blog update (`https://coreos.com/blog/making-sense-of-standards.html`), APPC and OCI will intersect only on runtime and APPC will continue to focus on image format, signing, and distribution. Runc is an implementation of OCI and Docker uses Runc.

Differences between Docker and Rkt

Following are some differences between Docker and Rkt Container runtimes:

- Docker uses LibContainer APIs to access the Linux kernel Container functionality while Rkt uses the Systemd-nspawn API to access the Linux kernel Container functionality. The following image illustrates this:



- Docker requires a daemon to manage Container images, remote APIs, and Container processes. Rkt is daemonless and Container resources are managed by systemd. This makes Rkt integrate better with init systems such as systemd and upstart.
- Docker has a complete platform to manage containers such as Machine, Compose, and Swarm. CoreOS will use some of its own tools such as Flannel for the Networking and combines it with tools such as Kubernetes for Orchestration.
- Docker is pretty mature and production-ready as compared to Rkt. As of the Rkt release 0.10.0, Rkt is not yet ready for production.
- For the Container image registry, Docker has the Docker hub and Rkt has Quay. Quay also has Docker images.

CoreOS is planning to support both Docker and Rkt and users will have a choice to use the corresponding Container runtime for their applications.

A workflow for distributed application development with Docker and CoreOS

The following is a typical workflow to develop microservices using Docker and CoreOS:

- Select applications that need to be containerized. This could be greenfield or legacy applications. For legacy applications, reverse engineering might be required to split the monolithic application and containerize the individual components.
- Create a Dockerfile for each microservice. The Dockerfile defines how to create the Container image from the base image. Dockerfile itself could be source-controlled.
- Split the stateless and stateful pieces of the application. For stateful applications, a storage strategy needs to be decided.
- Microservices need to talk to each other and some of the services should be reachable externally. Assuming that basic network connectivity between services is available, services can talk to each other either statically by defining a service name to IP address and port number mapping or by using service discovery where services can dynamically discover and talk to each other.
- Docker container images need to be stored in a private or public repository so that they can be shared among development, QA, and production teams.
- The application can be deployed in a private or public cloud. An appropriate infrastructure has to be selected based on the business need.
- Select the CoreOS cluster size and cluster architecture. It's better to make infrastructure dynamically scalable.
- Write CoreOS unit files for basic services such as etcd, fleet, and flannel.
- Finalize a storage strategy—local versus distributed versus cloud.
- For orchestration of smaller applications, fleet can be used. For complex applications, the Kubernetes kind of Orchestration solution will be necessary.
- For production clusters, appropriate monitoring, logging, and upgrading strategies also need to be worked out.

Summary

In this chapter, we covered the basics of CoreOS, Containers, and Docker and how they help in distributed application development and deployment. These technologies are under active development and will revolutionize and create a new software development and distribution model. We will explore each individual topic in detail in the following chapters. In the next chapter, we will cover how to set up the CoreOS development environment in Vagrant as well as in a public cloud.

References

- APPC specification: `https://github.com/appc/spec/blob/master/SPEC.md`
- OCI specification: `https://github.com/opencontainers/specs`
- CoreOS documentation: `https://coreos.com/docs/`
- Docker documentation: `https://docs.docker.com/`

Further reading and tutorials

- A blog on the minimalist operating system: `https://blog.docker.com/2015/02/the-new-minimalist-operating-systems/` and `https://blog.codeship.com/container-os-comparison/`
- Container basics: `http://www.slideshare.net/jpetazzo/anatomy-of-a-container-namespaces-cgroups-some-filesystem-magic-linuxcon`
- An introduction to Docker: `https://www.youtube.com/watch?v=Q5POuMHxW-0`
- Mesos overview: `https://www.youtube.com/watch?v=gVGZHzRjvo0`
- The CoreOS presentation: `http://www.slideshare.net/RichardLister/core-os`
- DigitalOcean CoreOS tutorials: `https://www.digitalocean.com/community/tags/coreos?type=tutorials`
- Microservices' characteristics: `http://martinfowler.com/articles/microservices.html`
- The Docker daemon issue: `http://www.ibuildthecloud.com/blog/2014/12/03/is-docker-fundamentally-flawed/` and `https://github.com/ibuildthecloud/systemd-docker`

2
Setting up the CoreOS Lab

CoreOS can be deployed in Bare Metal, VMs, or a cloud provider such as Amazon AWS or Google GCE. In this chapter, we will cover how to set up the CoreOS development environment in Vagrant, Amazon AWS, Google GCE, and Bare Metal. This development environment will be used in all the chapters going forward.

The following topics will be covered in this chapter:

- Cloud-config for CoreOS
- CoreOS with Vagrant
- CoreOS with Amazon AWS
- CoreOS with Google GCE
- The CoreOS installation on Bare Metal.
- The basic debugging of the CoreOS cluster

Different CoreOS deployment options are covered here because of the following reasons:

- Vagrant with Virtualbox is useful for users who don't have a cloud account.
- For some users, using a local machine might not be possible as VMs occupy a lot of resources, and using a cloud-based VM is the best choice in this case. As AWS and GCE are the most popular cloud providers, I chose these two.
- Bare metal installation would be preferable for traditional in-house data centers.
- In this book's examples, I have used one of the three approaches (Vagrant, AWS, and GCE) based on the simplicity of one of the approaches, better integration with one of the three approaches, or because of issues with a particular approach.

Cloud-config

Cloud-config is a declarative configuration file format that is used by many Linux distributions to describe the initial server configuration. The cloud-init program takes care of parsing `cloud-config` during server initialization and configures the server appropriately. The `cloud-config` file provides you with a default configuration for the CoreOS node.

The CoreOS cloud-config file format

The `coreos-cloudinit` program takes care of the default configuration of the CoreOS node during bootup using the `cloud-config` file. The `cloud-config` file describes the configuration in the YAML format (`http://www.yaml.org/`). CoreOS cloud-config follows the `cloud-config` specification with some CoreOS-specific options. The link, `https://coreos.com/os/docs/latest/cloud-config.html` covers the details of CoreOS `cloud-config`.

The main sections of cloud-config

The following are the main sections in the CoreOS `cloud-config` YAML file:

- CoreOS:
    - Etcd2: config parameters for etcd2
    - Fleet: config parameters for Fleet
    - Flannel: config parameters for Flannel
    - Locksmith: config parameters for Locksmith
    - Update: config parameters for automatic updates
    - Units: Systemd units that need to be started
- `ssh_authorized_keys`: Public keys for the *core* user
- `hostname`: Hostname for the CoreOS system
- `users`: Additional user account and group details
- `write_files`: Creates files with specified user data
- `manage_etc_hosts`: Specifies the contents of `/etc/hosts`

A sample CoreOS cloud-config

The following is a sample `cloud-config` file for a single node CoreOS cluster:

```
#cloud-config
coreos:
  etcd2:
    # Static cluster
    name: etcdserver
    initial-cluster-token: etcd-cluster-1
    initial-cluster: etcdserver=http://$private_ipv4:2380
    initial-cluster-state: new
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    # listen on both the official ports and the legacy ports
    # legacy ports can be omitted if your application doesn't depend
on them
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  fleet:
    public-ip: $public_ipv4
    metadata: "role=services"
  flannel:
    interface: $public_ipv4
  update:
      reboot-strategy: "etcd-lock"
  units:
    # To use etcd2, comment out the above service and uncomment these
    # Note: this requires a release that contains etcd2
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
    - name: flanneld.service
      drop-ins:
        - name: 50-network-config.conf
          content: |
            [Service]
            ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/
config '{ "Network": "10.1.0.0/16" }'
      command: start
    - name: docker-tcp.socket
      command: start
```

```
        enable: true
        content: |
          [Unit]
          Description=Docker Socket for the API
          [Socket]
          ListenStream=2375
          Service=docker.service
          BindIPv6Only=both
          [Install]
          WantedBy=sockets.target

write_files:
  - path: "/etc/motd"
    permissions: "0644"
    owner: "root"
    content: |
      --- My CoreOS Cluster ---
```

The following are some notes on the preceding `cloud-config`:

- The `etcd2` section specifies the configuration parameters for the `etcd2` service. In this case, we specify parameters needed to start `etcd` on the CoreOS node. The `public_ipv4` and `private_ipv4` environment variables are substituted with the CoreOS node's IP address. As there is only one node, we use the static cluster definition approach rather than using a discovery token. Based on the specified parameters, the `20-cloudinit.conf` Drop-In Unit gets created in `/run/systemd/system/etcd2.service.d` with the following environment variables:

```
[Service]
Environment="ETCD_ADVERTISE_CLIENT_URLS=http://172.17.8.101:2379"
Environment="ETCD_INITIAL_ADVERTISE_PEER_
URLS=http://172.17.8.101:2380"
Environment="ETCD_INITIAL_CLUSTER=etcdserver=ht
tp://172.17.8.101:2380"
Environment="ETCD_INITIAL_CLUSTER_STATE=new"
Environment="ETCD_INITIAL_CLUSTER_TOKEN=etcd-cluster-1"
Environment="ETCD_LISTEN_CLIENT_URLS=http://0.0.0.0:2379,ht
tp://0.0.0.0:4001"
Environment="ETCD_LISTEN_PEER_URLS=http://172.17.8.101:2380,ht
tp://172.17.8.101:7001"
Environment="ETCD_NAME=etcdserver"
```

- The `fleet` section specifies the configuration parameters for the `fleet` service, including any metadata for the node. The `20-cloudinit.conf` Drop-In Unit gets created in `/run/systemd/system/fleet.service.d` with the following environment variables:

```
[Service]
Environment="FLEET_METADATA=role=services"
Environment="FLEET_PUBLIC_IP=172.17.8.101"
```

- The update section specifies the update strategy for the CoreOS node. This gets updated in the node as `/etc/coreos/update.conf`:

```
GROUP=alpha
REBOOT_STRATEGY=etcd-lock
```

- The `units` section starts `etcd2`, `fleet`, and `flannel`. For `flannel`, we have a drop-in unit to update the subnet to be used for containers created with the Flannel network service. The `50-network-config.conf` Drop-in unit gets created in `/etc/systemd/system/flanneld.service.d`:

```
[Service]
ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config '{
"Network": "10.1.0.0/16" }'
```

- The `docker-tcp.socket` unit in the *units* section is a new `systemd` unit, and we specified the service content that allows for the docker daemon to be exposed through port `2375`. The unit will be created in `/etc/systemd/system`.
- The `write_files` section can be used to create any static files. An example could be a hello text when a user logs in, which we can do with `/etc/motd`. The hello message would look as follows:

```
Last login: Tue Sep 15 14:15:04 2015 from 10.0.2.2
--- My CoreOS Cluster ---
core@core01 ~ $
```

The cloud-config validator

Cloud-config uses the YAML syntax. YAML is a human-readable data serialization format and uses indents and spaces for alignment. It is better to validate the `cloud-config` YAML configuration files before using them. There are two options to validate the CoreOS `cloud-config`.

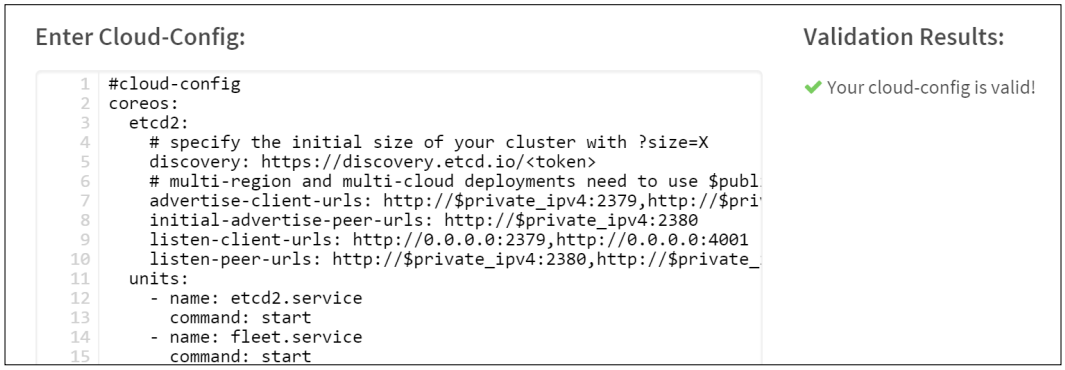
A hosted validator

Use this CoreOS-provided link (`https://coreos.com/validate/`) to validate `cloud-config`.

Here is an example of a valid and invalid `cloud-config` and the results using the validator.

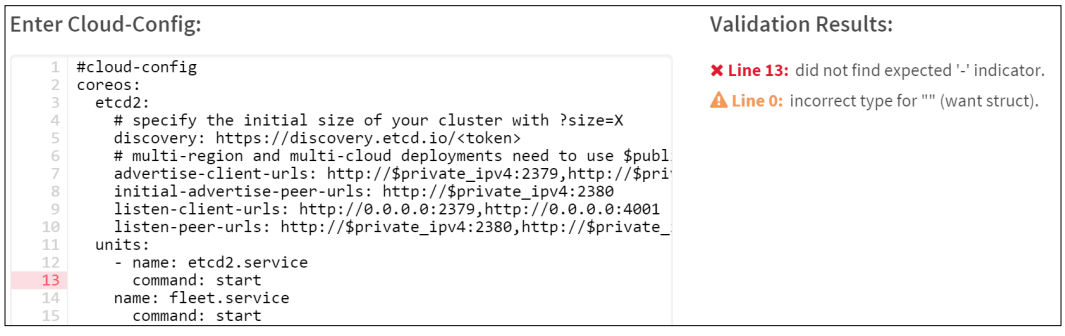
Valid cloud-config

As we can see in the following screenshot, the validator says that the following `cloud-config` is valid:



Invalid cloud-config

Here, we can see that the validator has specified that `-` is missing in line 14. YAML uses spaces for the delimiting, so we need to make sure that the number of spaces is exact:



The cloudinit validator

We can use the `coreos-cloudinit --validate` option available in CoreOS to validate the cloud-config. Let's look at the following sample `cloud-config`:

```
#cloud-config

coreos:
  units:
    - name: docker-tcp.socket
      command: start
      enable: yes
      content: |
        [Unit]
        Description=Docker Socket for the API

        [Socket]
        ListenStream=2375
        BindIPv6Only=both
        Service=docker.service

        [Install]
        WantedBy=sockets.target
```

When we validate this, we get no errors, as shown in the following screenshot:

```
core@core01 ~ $ coreos-cloudinit --validate --from-file="sample-cloud-config"
2015/09/15 16:42:39 Checking availability of "local-file"
2015/09/15 16:42:39 Fetching user-data from datasource of type "local-file"
```

Now, let's try the same `cloud-config` with errors. Here, we have `|` missing in the content line:

```
#cloud-config

coreos:
  units:
    - name: docker-tcp.socket
      command: start
      enable: yes
      content:
        [Unit]
        Description=Docker Socket for the API

        [Socket]
        ListenStream=2375
        BindIPv6Only=both
        Service=docker.service

        [Install]
        WantedBy=sockets.target
```

We see the following errors when we validate:

```
core@core01 ~ $ coreos-cloudinit --validate --from-file="sample-cloud-config"
2015/09/15 16:44:18 Checking availability of "local-file"
2015/09/15 16:44:18 Fetching user-data from datasource of type "local-file"
2015/09/15 16:44:18 line 9: error: did not find expected key
2015/09/15 16:44:18 line 0: warning: incorrect type for "" (want struct)
```

Executing cloud-config

There are two `cloud-config` files that are run as part of the CoreOS bootup:

- System cloud-config
- User cloud-config

System `cloud-config` is given by the provider (such as Vagrant or AWS) and is embedded as part of the CoreOS provider image. Different providers such as Vagrant, AWS, and GCE have their `cloud-config` present in `/usr/share/oem/cloud-config.yaml`. This `cloud-config` is responsible for setting up the provider-specific configurations, such as networking, SSH keys, mount options, and so on. The `coreos-cloudinit` program first executes system `cloud-config` and then user `cloud-config`.

Depending on the provider, user `cloud-config` can be supplied using either config-drive or an internal user data service. Config-drive is a universal way to provide `cloud-config` by mounting a read-only partition that contains `cloud-config` to the host machine. Rackspace uses config-drive to get user `cloud-config`, and AWS uses its internal user data service to fetch the user data and doesn't rely on config-drive. In the Vagrant scenario, Vagrantfile takes care of copying the `cloud-config` to the CoreOS VM.

The CoreOS cluster with Vagrant

Vagrant can be installed in Windows or Linux. The following is my development environment for the Vagrant CoreOS:

- Windows 7: I use mysysgit (`https://git-for-windows.github.io/`) to get a Linux-like shell for Windows
- Vagrant 1.7.2: `https://www.vagrantup.com/downloads.html`
- Virtualbox 4.3.28: `https://www.virtualbox.org/wiki/Downloads`

For a few of the examples in the book, I have used Vagrant to run CoreOS inside a Linux VM running on top of Windows laptop with Virtualbox.

Steps to start the Vagrant environment

1. Check out the coreos-vagrant code base:

   ```
   git clone https://github.com/coreos/coreos-vagrant.git
   ```

2. Copy the sample `user-data` and `config.rb` files in the coreos-vagrant directory:

   ```
   cd coreos-vagrant
   mv user-data.sample user-data
   mv config.rb.sample config.rb
   ```

3. Edit `Vagrantfile`, `user-data`, and `config.rb` based on your need.
4. Start the CoreOS cluster:

   ```
   Vagrant up
   ```

5. SSH to the individual node:

   ```
   Vagrant ssh core-<id>
   ```

Important files to be modified

The following are important files to be modified along with commonly needed modifications.

Vagrantfile

Vagrant sets up the VM environment based on the configuration defined in `Vagrantfile`. The following are certain relevant functionalities in the CoreOS context:

- The version of CoreOS software to be used is specified using `update_channel`. The version can be specified as `stable`, `beta`, and `alpha`. More details on CoreOS software versions are covered in *Chapter 3, CoreOS Autoupdate*.
- CPU and memory for the VM and ports to be exposed from the VM.
- SSH key management.

User-data

The `user-data` file is essentially the `cloud-config` file that specifies the discovery token, environment variables, and list of units to be started by default. Vagrant copies the `cloud-config` file to `/var/lib/coreos-vagrant/vagrantfile-user-data` inside the VM. The `coreos-cloudinit` reads `vagrantfile-user-data` on every boot and uses it to create the machine's user data file.

Config.rb

The `config.rb` file specifies the count of CoreOS nodes. This file also provides you with an option to automatically generate a discovery token. Some options here overlap with the `Vagrantfile` like image version.

Vagrant – a three-node cluster with dynamic discovery

Here, we will create a three-node CoreOS cluster with `etcd2` and `fleet` running on each node and nodes discovering each other dynamically.

Generating a discovery token

When we start a multinode CoreOS cluster, there needs to be a bootstrapping mechanism to discover the cluster members. For this, we generate a token specifying the number of initial nodes in the cluster as an argument. Each node needs to be started with this discovery token. Etcd will use the discovery token to put all the nodes with the same discovery token as part of the initial cluster. CoreOS runs the service to provide the discovery token from its central servers.

There are two approaches to generate a discovery token:

From the browser: `https://discovery.etcd.io/new?size=3`

Using curl: `curl https://discovery.etcd.io/new?size=3`

The following is a curl example with a generated discovery token. This token needs to be copied to `user-data`:

```
$ curl https://discovery.etcd.io/new?size=3
https://discovery.etcd.io/9a6b7af06c8a677b4e5f76ae9ce0da9c
```

Steps for cluster creation

The following is a `cloud-config` user data with the updated discovery token that we generated in the preceding section along with the necessary environment variables and service units. All three nodes will use this `cloud-config`:

```
#cloud-config
coreos:
  etcd2:
    #generate a new token for each unique cluster from https://
discovery.etcd.io/new
```

```
    discovery: https://discovery.etcd.io/9a6b7af06c8a677b4e5f76ae9ce0
da9c
    # multi-region and multi-cloud deployments need to use $public_
ipv4
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    # listen on both the official ports and the legacy ports
    # legacy ports can be omitted if your application doesn't depend
on them
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  fleet:
    public-ip: $public_ipv4
  flannel:
    interface: $public_ipv4
  units:
    # Note: this requires a release that contains etcd2
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

We need to update `num_instances` to `3` in `config.rb` and then perform `vagrant up`.

To verify the basic cluster operation, we can check the following output, where we should see the cluster members.

The following `etcdctl` member output shows the three cluster members:

```
core@core-01 ~ $ etcdctl member list
f6c11cf0535a1e6: name=f3b106e2bb314703934d55820cd769d3 peerURLs=http://172.17.8.103:2380 clientURLs=http://172.17.8.103:2379
2a4f758b88419fd2: name=99d20f9acdb547d8af77e466e23c95a9 peerURLs=http://172.17.8.102:2380 clientURLs=http://172.17.8.102:2379
66f1dc236b951188: name=149674dd09c240b3a43bdb3ad5678189 peerURLs=http://172.17.8.101:2380 clientURLs=http://172.17.8.101:2379
```

The following fleet member output shows the three cluster members:

```
core@core-01 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
149674dd...     172.17.8.101    -
99d20f9a...     172.17.8.102    -
f3b106e2...     172.17.8.103    -
```

Vagrant – a three-node cluster with static discovery

Here, we will create a three-node CoreOS cluster and use a static approach to mention its cluster neighbors. In the dynamic discovery approach, we need to use a discovery token to discover the cluster members. Static discovery can be used for scenarios where access to the token server is not available to cluster members, and the cluster member IP addresses are known in advance.

Perform the following steps:

1. First, we need to create three separate instances of the CoreOS Vagrant environment by performing `git clone` separately for each node.
2. The `config.rb` file must be updated for each node with `num_instances` set to one.
3. Vagrantfile should be updated for each node so that IP addresses are statically assigned as `172.17.8.101` for `core-01`, `172.17.8.102` for `core-02`, and `172.17.8.103` for `core-03`. IP addresses should be updated based on your environment.

The `cloud-config` user data for the first node is as follows:

```
#cloud-config
coreos:
  etcd2:
    name: core-01
    initial-advertise-peer-urls: http://172.17.8.101:2380
    listen-peer-urls: http://172.17.8.101:2380
    listen-client-urls: http://172.17.8.101:2379,http://127.0.0.1:2379
    advertise-client-urls: http://172.17.8.101:2379
    initial-cluster-token: etcd-cluster-1
    initial-cluster: core-01=http://172.17.8.101:2380,core-
02=http://172.17.8.102:2380,core-03=http://172.17.8.103:2380
    initial-cluster-state: new
  fleet:
    public-ip: $public_ipv4
  flannel:
    interface: $public_ipv4
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

The `cloud-config` user data for the second node is as follows:

```
#cloud-config
coreos:
  etcd2:
    name: core-02
    initial-advertise-peer-urls: http://172.17.8.102:2380
    listen-peer-urls: http://172.17.8.102:2380
    listen-client-urls: http://172.17.8.102:2379,http://127.0.0.1:2379
    advertise-client-urls: http://172.17.8.102:2379
    initial-cluster-token: etcd-cluster-1
    initial-cluster: core-01=http://172.17.8.101:2380,core-
02=http://172.17.8.102:2380,core-03=http://172.17.8.103:2380
    initial-cluster-state: new
  fleet:
    public-ip: $public_ipv4
  flannel:
    interface: $public_ipv4
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

The `cloud-config` user data for the third node is as follows:

```
#cloud-config
coreos:
  etcd2:
    name: core-03
    initial-advertise-peer-urls: http://172.17.8.103:2380
    listen-peer-urls: http://172.17.8.103:2380
    listen-client-urls: http://172.17.8.103:2379,http://127.0.0.1:2379
    advertise-client-urls: http://172.17.8.103:2379
    initial-cluster-token: etcd-cluster-1
    initial-cluster: core-01=http://172.17.8.101:2380,core-
02=http://172.17.8.102:2380,core-03=http://172.17.8.103:2380
    initial-cluster-state: new
  fleet:
    public-ip: $public_ipv4
  flannel:
    interface: $public_ipv4
  units:
```

```
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

We need to perform `vagrant up` separately for each of the nodes. We should see the cluster member list updated in both the `etcdctl member list` and `fleetctl list-machines` outputs.

Vagrant – a production cluster with three master nodes and three worker nodes

In *Chapter 1*, *CoreOS Overview*, we covered the CoreOS cluster architecture. A production cluster has one set of nodes (called **master**) to run critical services, and another set of nodes (called **worker**) to run application services. In this example, we create three master nodes running `etcd` and other critical services and another three worker nodes. Etcd in the worker nodes will proxy to the master nodes. Worker nodes will be used for user-created services while master nodes will be used for system services. This avoids resource contention. The following are the steps needed for this creation:

- Create a Vagrant three-node cluster for the master and a three-node cluster for the worker.
- Update Vagrantfile to use non-conflicting IP address ranges between the master and worker nodes.
- Use the dynamic discovery token approach to create a token for the three-node clusters and update the `cloud-config` user data for both the master and worker nodes to the same token. We have specified the token size as `3` as worker nodes don't run `etcd`.

The following is the user data for the master cluster:

```
#cloud-config
coreos:
  etcd2:
    discovery: https://discovery.etcd.io/
d49bac8527395e2a7346e694124c8222
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    # listen on both the official ports and the legacy ports
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
```

```
  fleet:
     metadata: "role=master"
     public-ip: $public_ipv4
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

The following is the user data for the worker cluster. The discovery token needs to be the same for the master and worker clusters:

```
#cloud-config
coreos:
  etcd2:
    discovery: https://discovery.etcd.io/
d49bac8527395e2a7346e694124c8222
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    # listen on both the official ports and the legacy ports
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  fleet:
     metadata: "role=worker"
     public-ip: $public_ipv4
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

The only difference between the master and worker user data is in the metadata used for fleet. In this example, we used `role` as `master` for the master cluster and `role` as `worker` for the worker cluster.

Let's look at the `etcdctl` member list and fleet machine list. The following output will be the same across all the nodes in the master and worker cluster.

The `etcdctl` member output is as follows:

```
core@core-01 ~ $ etcdctl member list
453db3e7ac59db7a: name=e3201b73095048778045dab8470136a3 peerURLs=http://172.17.8.100:2380 clientURLs=http://172.17.8.100:2379
5746a3aa353c0d8a: name=e59c9e194ec144c389d4dc49213a6b4d peerURLs=http://172.17.8.102:2380 clientURLs=http://172.17.8.102:2379
5a6c69ee0972c229: name=79d0e188870d400fb30d7e8c17431bc9 peerURLs=http://172.17.8.101:2380 clientURLs=http://172.17.8.101:2379
```

The fleet member output is as follows:

```
core@core-01 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
3e0ccf1a...     172.17.8.107    role=worker
64031232...     172.17.8.108    role=worker
79d0e188...     172.17.8.101    role=master
7d6cdbc9...     172.17.8.106    role=worker
e3201b73...     172.17.8.100    role=master
e59c9e19...     172.17.8.102    role=master
```

The following is the `journalctl -u etcd2.service` output on worker nodes that show worker nodes proxying to master nodes:

```
Sep 17 06:12:24 corew-01 etcd2[1040]: 2015/09/17 06:12:24 etcdmain: discovery cluster full, falling back to proxy
Sep 17 06:12:25 corew-01 etcd2[1040]: 2015/09/17 06:12:25 etcdmain: proxy: using peer urls [http://172.17.8.100:2380 http://172.17.8.101:2380 h
ttp://172.17.8.102:2380]
Sep 17 06:12:25 corew-01 etcd2[1040]: 2015/09/17 06:12:25 etcdmain: proxy: listening for client requests on 0.0.0.0:2379
Sep 17 06:12:25 corew-01 etcd2[1040]: 2015/09/17 06:12:25 etcdmain: proxy: listening for client requests on 0.0.0.0:4001
```

A CoreOS cluster with AWS

Amazon AWS provides you with a public cloud service. CoreOS can be run on the VMs provided by AWS. The following are some prerequisites for this setup:

- You need an account in AWS. AWS provides you with a one-year trial account for free.
- Create and download a key pair. The key pair is needed to SSH to the nodes.
- The AWS interface can be accessed through the AWS console, which is a GUI interface, or by AWS CLI. AWS CLI (`http://aws.amazon.com/cli/`) can be installed in either Windows or Linux.

The following are two approaches of creating a CoreOS cluster with AWS.

AWS – a three-node cluster using Cloudformation

Cloudformation is an AWS orchestration tool to manage a collection of AWS resources that include compute, storage, and networking. The link, `https://s3.amazonaws.com/coreos.com/dist/aws/coreos-stable-hvm.template`, has the template file for the CoreOS cluster. The following are some of the key sections in the template:

- The AMI image ID to be used based on the region
- The EC2 Instance type

- The security group configuration
- The CoreOS cluster size including the minimum and maximum size to autoscale
- The initial cloud-config to be used

For the following example, I modified the template to use `t2.micro` instead of `m3.medium` for the instance size. The following CLI can be used to create a three-node CoreOS cluster using `Cloudformation`. The discovery token in the below command needs to be updated with the generated token for your case:

```
aws cloudformation create-stack \
    --stack-name coreos-test \
    --template-body file://mycoreos-stable-hvm.template \
    --capabilities CAPABILITY_IAM \
    --tags Key=Name,Value=CoreOS \
    --parameters \      ParameterKey=DiscoveryURL,ParameterValue="htt
ps://discovery.etcd.io/925755234ab82c1ef7bcfbbacdd8c088" \
        ParameterKey=KeyPair,ParameterValue="keyname"
```

The following is the output of the successful stack using `aws cloudformation list-stacks`:

```
"StackSummaries": [
   {
       "StackId": "arn:aws:cloudformation:us-west-2:173760706945:stack/coreos-test/d5c35b10-5d43-11e5-af9e-50fa5e75180a",
       "StackName": "coreos-test",
       "CreationTime": "2015-09-17T13:56:03.534Z",
       "StackStatus": "CREATE_COMPLETE",
       "TemplateDescription": "CoreOS on EC2: http://coreos.com/docs/running-coreos/cloud-providers/ec2/"
   },
```

After the preceding step, we can see that members are getting discovered successfully by both `etcd` and `fleet`.

AWS – a three-node cluster using AWS CLI

The following are some prerequisites to create a CoreOS cluster in AWS using AWS CLI:

1. Create a token for a three-node cluster from the discovery token service.
2. Set up a security group exposing the ports `ssh`, `icmp`, `2379`, and `2380`. `2379` and `2380` are needed for the `etcd2` client-to-server and server-to-server communication.
3. Determine the AMI image ID using this link (`https://coreos.com/os/docs/latest/booting-on-ec2.html`) based on your AWS Zone and update channel. The latest image IDs for different AWS Zones get automatically updated in this link.

The following CLI will create the three-node cluster:

```
aws ec2 run-instances --image-id ami-85ada4b5 --count 3 --instance-type
t2.micro --key-name "yourkey" --security-groups "coreos-test" --user-data
file://cloud-config.yaml
```

Here, the `ami-85ada4b5` image ID is from the stable update channel. The `coreos-test` security group has the necessary ports that need to be exposed outside.

The following is the `cloud-config` that I used:

```
#cloud-config
coreos:
  etcd2:
    # specify the initial size of your cluster with ?size=X
    discovery: https://discovery.etcd.io/47460367c9b15edffeb49de30c
ab9354
    advertise-client-urls: http://$private_ipv4:2379,http://$private_
ipv4:4001
    initial-advertise-peer-urls: http://$private_ipv4:2380
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

The following output shows the `etcd` member list and fleet member list with three nodes in the cluster:

```
core@ip-172-31-38-139 ~ $ etcdctl member list
315e6f582d93971c: name=63e51b3dd585486d905fe8e64de5e462 peerURLs=http://172.31.38.139:2380 clientURLs=http://172.31.38.139:2379,http://172.31.38.139:4001
44228d6d8ff4f3ff: name=2e6e21c139114d5c90c061c5617c8eef peerURLs=http://172.31.38.138:2380 clientURLs=http://172.31.38.138:2379,http://172.31.38.138:4001
831e27eeed6546ad: name=d29432a15745407388444d585a5c554a peerURLs=http://172.31.38.137:2380 clientURLs=http://172.31.38.137:2379,http://172.31.38.137:4001
core@ip-172-31-38-139 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
2e6e21c1...     172.31.38.138   -
63e51b3d...     172.31.38.139   -
d29432a1...     172.31.38.137   -
```

The same example can be tried from the AWS Console, where we can specify the options from the GUI.

A CoreOS cluster with GCE

Google's GCE is another public cloud provider like Amazon AWS. CoreOS can be run on the VMs provided by GCE. The following are some prerequisites for this setup:

- You need a GCE account. GCE provides you with a free trial account for 60 days.
- GCE resources can be accessed using gcloud SDK or GCE GUI Console. SDK can be downloaded from `https://cloud.google.com/sdk/`.
- A base project in GCE needs to be created under which all the resources reside.
- A security token needs to be created, which is used for SSH access.

GCE – a three-node cluster using GCE CLI

The following are some prerequisites to create a CoreOS cluster in GCE:

- Create a token for a three-node cluster from a discovery token service.
- Set up a security group exposing the ports `ssh`, `icmp`, `2379`, and `2380`. `2379` and `2380` are needed for the `etcd2` client-to-server and server-to-server communication.
- The link, `https://coreos.com/os/docs/latest/booting-on-google-compute-engine.html`, gets automatically updated with the latest GCE CoreOS releases from the stable, beta, and alpha channels. We need to pick the appropriate image that is needed.

The following CLI can be used to create a three-node CoreOS GCE cluster from the stable release:

```
gcloud compute instances create core1 core2 core3 --image https://
www.googleapis.com/compute/v1/projects/coreos-cloud/global/images/
coreos-stable-717-3-0-v20150710 --zone us-central1-a --machine-type n1-
standard-1 --metadata-from-file user-data=cloud-config.yaml
```

The following is the `cloud-config.yaml` file that's used:

```
#cloud-config
coreos:
  etcd2:
    # specify the initial size of your cluster with ?size=X
    discovery: https://discovery.etcd.io/46ad006905f767331a36bb2a4dbd
e3f5
```

```
    advertise-client-urls: http://$private_ipv4:2379,http://$private_
ipv4:4001
    initial-advertise-peer-urls: http://$private_ipv4:2380
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

We can SSH to any of the nodes using `gcloud compute ssh <nodeid>`.

The following output shows you that the cluster is created successfully and members are seen from both `etcd` and `fleet`:

```
core@core1 ~ $ etcdctl member list
14ac808c38008fb7: name=8ca8be04b8640573195e733c402cccad peerURLs=http://10.240.82.102:2380 clientURLs=http://10.240.82.102:2379,http://10.240.82.102:4001
600a6adb8487263a: name=e93455e373efb5ccb8d3d7b0702b6e58 peerURLs=http://10.240.250.187:2380 clientURLs=http://10.240.250.187:2379,http://10.240.250.187:4001
6221bd73940e0ca3: name=b17227b7a289e98226caa0d232c61158 peerURLs=http://10.240.42.201:2380 clientURLs=http://10.240.42.201:2379,http://10.240.42.201:4001
core@core1 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
8ca8be04...     10.240.82.102   -
b17227b7...     10.240.42.201   -
e93455e3...     10.240.250.187  -
```

The CoreOS cluster can also be created using the GCE Console GUI interface.

CoreOS installation on Bare Metal

There are two approaches to install CoreOS on Bare Metal:

- CoreOS ISO image
- PXE or IPXE boot

The steps below covers the approach to install CoreOS on Bare Metal using an ISO image.

I installed using a CoreOS ISO image on the Virtualbox CD drive. The procedure should be the same if we burn the ISO image on CD and then install on Bare Metal.

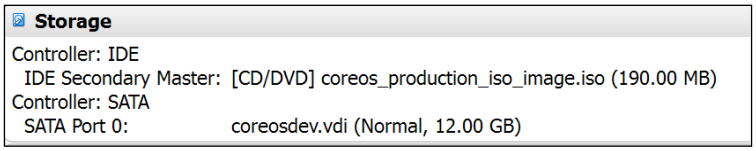
The following is summary of the steps:

1. Download the required ISO image based on stable, beta, and alpha versions from `https://coreos.com/os/docs/latest/booting-with-iso.html`.
2. Start a new Linux machine in Virtualbox with the required CPU, memory, and network settings, and mount the ISO image on the IDE drive.

3. Create an SSH key to log in to the CoreOS node using `ssh-keygen`.
4. Start the Linux machine and then use the CoreOS script to install CoreOS on the hard disk with the necessary `cloud-config`. The cloud-config used here is similar to cloud-config is used in previous sections, SSH key needs to be manually updated.
5. Remove the CD drive from Virtualbox and reboot. This will load the CoreOS image from the hard disk.

I have used the stable ISO image version 766.4.0.

The following screenshot shows you the initial storage mounting on Virtualbox with the ISO image on the IDE drive:



The easiest way to get `cloud-config` is by `wget`. When we boot from the CD, we cannot cut and paste as there is no Windows manager. The easiest way to get `cloud-config` to the node is by having cloud-config in a hosting location and fetch it using `wget`. The SSH key needs to be updated appropriately.

```
wget https://github.com/smakam/coreos/raw/master/single-node-cloudconfig.yml
```

The installation of CoreOS to the hard disk can be done using the CoreOS-provided script:

```
sudo coreos-install -d /dev/sda -C stable -c ~/cloud-config.yaml
```

After successful installation, we can shut down the node and remove the IDE drive so that the bootup can happen from the hard disk. The following screenshot shows you the storage selection in Virtualbox to boot using the hard disk:

After the node is booted up, we can SSH to the node as we have already set up the SSH key. The following output shows you the CoreOS version on Bare Metal:

```
core@ip-172-31-26-79 ~ $ fleetctl ssh b5ce6ddfe76243789dd742d5f18fd052 cat /etc/machine-id
b5ce6ddfe76243789dd742d5f18fd052
```

Basic debugging

The following are some basic debugging tools and approaches to debug issues in the CoreOS cluster.

journalctl

Systemd-Journal takes care of logging all the kernel and systemd services. Journal log files from all the services are stored in a centralized location in `/var/log/journal`. The logs are stored in the binary format, and this keeps it easy to manipulate to different formats.

Here are some common examples that shows how to use Journalctl:

- `Journalctl`: This lists the combined journal log from all the sources.
- `Journalctl -u etcd2.service`: This lists the logs from `etcd2.service`.
- `Journalctl -u etcd2.service -f`: This lists the logs from `etcd2`. service like `tail -f` format.
- `Journalctl -u etcd2.service -n 100`: This lists the logs of the last 100 lines.
- `Journalctl -u etcd2.service -no-pager`: This lists the logs with no pagination, which is useful for search.
- `Journalctl -p err -n 100`: This lists all 100 errors by filtering the logs.
- `journalctl -u etcd2.service --since today`: This lists today's logs of `etcd2.service`.
- `journalctl -u etcd2.service -o json-pretty`: This lists the logs of `etcd2.service` in JSON-formatted output.

systemctl

The `systemctl` utility can be used for basic monitoring and troubleshooting of the systemd units.

The following example shows you the status of the `systemdunit docker.service`:

```
core@core-01 ~ $ systemctl status docker.service
? docker.service - Docker Application Container Engine
   Loaded: loaded (/usr/lib64/systemd/system/docker.service; disabled; vendor preset: disabled)
   Active: active (running) since Thu 2015-09-17 14:43:21 UTC; 3s ago
     Docs: http://docs.docker.com
 Main PID: 5493 (docker)
   Memory: 2.7M
      CPU: 65ms
   CGroup: /system.slice/docker.service
           └─5493 docker --daemon --host=fd://
```

We can stop and restart services in case there are issues with a particular service.

The following command will restart docker.service:

```
sudo systemctl restart docker.service
```

When a service file is changed or environment variables are changed, we need to execute the following command to reload configuration before restarting the service for the changes to take effect:

```
sudo systemctl daemon-reload
```

The following command is useful to see the units that have failed:

```
Systemctl --failed
```

Cloud-config

Earlier, we looked at how the `cloud-config` YAML file can be prevalidated. In case there are runtime errors, we can check it with `journalctl -b _EXE=/usr/bin/coreos-cloudinit`.

If we make changes to the `cloud-config` user data after the initial node setup, we can perform the following steps to activate the new configuration:

- Perform `vagrant reload --provision` to get the new configuration.
- The new `cloud-config` user data will be in `/var/lib/coreos-vagrant` as `vagrantfile-user-data`. Perform `sudo coreos-cloudinit --from-file vagrantfile-user-data` to update the new configuration.

Logging from one CoreOS node to another

Sometimes, it is useful to SSH to other nodes from one of the CoreOS nodes in the cluster. The following set of commands can be used to forward the SSH agent that we can use to SSH from other nodes. More information on SSH agent forwarding can be found at `http://rabexc.org/posts/using-ssh-agent`.

```
eval `ssh-agent`
ssh-add <key> (Key is the private key)
ssh -i <key> core@<ip> -A (key is the private key)
```

After this, we can either SSH to the machine ID or a specific Fleet unit, as shown in the following screenshot:

```
core@ip-172-31-26-79 ~ $ fleetctl ssh b5ce6ddfe76243789dd742d5f18fd052 cat /etc/machine-id
b5ce6ddfe76243789dd742d5f18fd052
```

Note: SSH agent forwarding is not secure and should be used only to debug.

Important files and directories

Knowing these files and directories helps with debugging the issues:

- Systemd unit file location - `/usr/lib64/systemd/system`.
- Network unit files - `/usr/lib64/systemd/network`.
- User-written unit files and drop-ins to change the default parameters - `/etc/systemd/system`. Drop-ins for specific configuration changes can be done using the configuration file under the specific service directory. For example, to modify the fleet configuration, create the `fleet.service.d` directory and put the configuration file in this directory.
- User-written network unit files - `/etc/systemd/network`.
- Runtime environment variables and drop-in configuration of individual components such as `etcd` and `fleet` - `/run/systemd/system/`.
- The vagrantfile user data containing the `cloud-config` user data used with Vagrant - `/var/lib/coreos-vagrant`.
- The `systemd-journald` logs - `/var/log/journal`.
- `cloud-config.yaml` associated with providers such as Vagrant, AWS, and `GCE`- `/usr/share/oem`. (CoreOS first executes this `cloud-config` and then executes the user-provided `cloud-config`.)

- Release channel and update strategy - `/etc/coreos/update.conf`.
- The public and private IP address (`COREOS_PUBLIC_IPV4` and `COREOS_PRIVATE_IPV4`) - `/etc/environment`.
- The machine ID for the particular CoreOS node - `/etc/machine-id`.
- The flannel network configuration - `/run/flannel/`.

Common mistakes and possible solutions

- For CoreOS on the cloud provider, there is a need to open up ports 2379 and 2380 on the VM. 2379 is used for etcd client-to-server communication, and 2380 is used for etcd server-to-server communication.
- A discovery token needs to be generated every time for each cluster and cannot be shared. When a stale discovery token is shared, members will not be able to join the etcd cluster.
- Running multiple CoreOS clusters with Vagrant simultaneously can cause issues because of overlapping IP ranges. Care should be taken so that common parameters such as the IP address are not shared across clusters.
- Cloud-config YAML files need to be properly indented. It is better to use the cloud-config validator to check for issues.
- When using discovery token, CoreOS node needs to have Internet access to access the token service.
- When creating a discovery token, you need to use the size based on the count of members and all members need to be part of the bootstrap. If all members are not present, the cluster will not be formed. Members can be added or removed later.

Summary

In this chapter, we covered the basics of CoreOS cloud-config and how to set up the CoreOS development environment with Vagrant, Amazon AWS, Google GCE, and Bare Metal. We also covered some basic debugging steps for commonly encountered issues. As described in this chapter, it is easy to install CoreOS in the local data center or Cloud environments. It is better to try out deployment in a development cluster before moving to production environments. In the next chapter, we will cover how the CoreOS automatic update works.

References

- Vagrant installation: `https://coreos.com/os/docs/latest/booting-on-vagrant.html`
- AWS installation: `https://coreos.com/os/docs/latest/booting-on-ec2.html`
- GCE installation: `https://coreos.com/os/docs/latest/booting-on-google-compute-engine.html`
- Bare Metal installation: `https://coreos.com/os/docs/latest/installing-to-disk.html`
- CoreOS CloudInit: `https://github.com/coreos/coreos-cloudinit`

Further reading and tutorials

- Introduction to the cloud-config format: `https://www.digitalocean.com/community/tutorials/an-introduction-to-cloud-config-scripting`
- CoreOS with AWS Cloudformation: `http://blog.michaelhamrah.com/2015/03/managing-coreos-clusters-on-aws-with-cloudformation/`
- The CoreOS bare-metal installation: `http://stevieholdway.tumblr.com/post/90167512059/coreos-bare-metal-iso-install-tutorial` and `http://linuxconfig.org/how-to-perform-a-bare-metal-installation-of-coreos-linux`
- Using journalctl to view systemd logs: `https://www.digitalocean.com/community/tutorials/how-to-use-journalctl-to-view-and-manipulate-systemd-logs`

3
CoreOS Autoupdate

One of the missions of CoreOS is to keep the operating system as secure as possible. One way to achieve this is to keep the OS up to date with the latest patches. The CoreOS automatic update scheme provides you with a secure, reliable, and robust mechanism that provides pushed updates. CoreOS provides enough controls to the user to control the update based on their environment.

This chapter will cover the following topics:

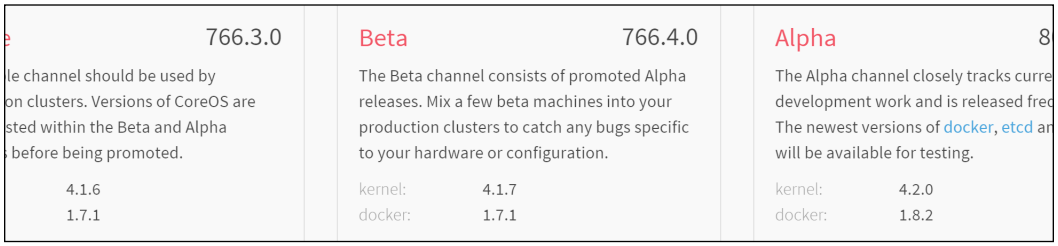
- The CoreOS release cycle
- The partition scheme used in CoreOS
- The CoreOS automatic update infrastructure
- The configuration of the CoreOS update
- The CoreUpdate commercial service from CoreOS

All examples from this chapter will use CoreOS in the AWS environment. There is a section on *Vagrant CoreOS update* where Vagrant-specific CoreOS updates are mentioned.

The CoreOS release cycle

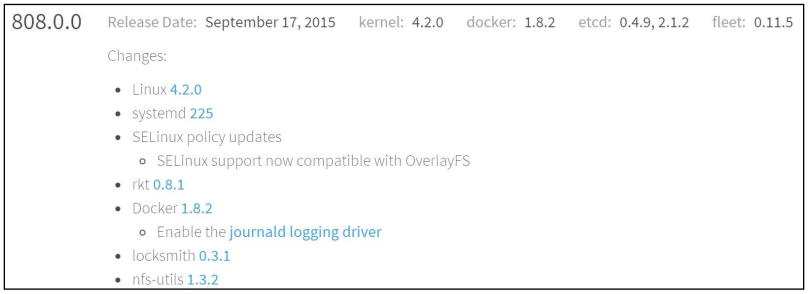
Alpha, Beta, and Stable are release channels within CoreOS. CoreOS releases progress through each channel in this order: Alpha->Beta->Stable. An Alpha channel is a development channel. An Alpha release in the Alpha channel gets promoted to the Beta channel after reaching defined quality level and becomes a Beta release. A Beta release in the Beta channel gets promoted to the Stable channel when it gets to production quality and becomes a Stable release. All releases get started as Alpha, but the promotion to Beta and Stable happens on the basis of testing.

The CoreOS release page reflects the latest Alpha, Beta, and Stable releases (`https://coreos.com/releases/`). The following are the latest releases as of August 19, 2015:



The major version number (for example, **766** in **766.3.0**) is the number of days from July 13, 2013, which was the CoreOS epoch.

As CoreOS is composed of multiple system components such as etcd, fleet, flannel, Docker, and RKT, every release will have a particular version of the system components based on the stability of individual components. For example, the following are the versions of the critical system components as of CoreOS version **808.0.0**:



The following command can be used to check the CoreOS version in the node. The node here is running image `723.3.0`, which was a stable release at that point:

```
core@core-01 ~ $ cat /etc/os-release
NAME=CoreOS
ID=coreos
VERSION=723.3.0
VERSION_ID=723.3.0
BUILD_ID=
PRETTY_NAME="CoreOS 723.3.0"
ANSI_COLOR="1;32"
HOME_URL="https://coreos.com/"
BUG_REPORT_URL="https://github.com/coreos/bugs/issues"
```

The following command can be used to check the CoreOS Linux kernel version:

```
core@core-01 ~ $ uname -a
Linux core-01 4.0.5 #2 SMP Fri Jul 10 01:01:50 UTC 2015 x86_64 Intel(R) Core(TM) i7-4800MQ CPU @ 2.70GHz GenuineIntel GNU/Linux
```

The following are versions of critical system components in CoreOS release 723.3.0:

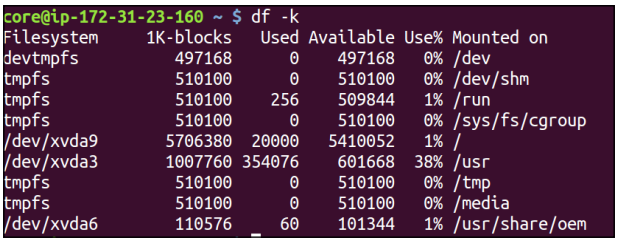
```
core@ip-172-31-39-192 ~ $ etcd2 --version
etcd Version: 2.1.2
Git SHA: ff8d1ec
Go Version: go1.4.2
Go OS/Arch: linux/amd64
core@ip-172-31-39-192 ~ $ fleet --version
fleetd version 0.10.2
core@ip-172-31-39-192 ~ $ docker --version
Docker version 1.7.1, build 2c2c52b-dirty
core@ip-172-31-39-192 ~ $ rkt version
rkt version 0.7.0
appc version 0.6.1
```

The partition table on CoreOS

The partition table shows you the disk partitions maintained by the OS. The following image shows you a partition table in one of the CoreOS cluster nodes using the `sudo cgpt show /dev/xvda` command:

```
core@ip-172-31-23-160 ~ $ sudo cgpt show /dev/xvda
      start        size    part  contents
          0           1          Hybrid MBR
          1           1          Pri GPT header
          2          32          Pri GPT table
       4096      262144       1  Label: "EFI-SYSTEM"
                                   Type: EFI System Partition
                                   UUID: FACA3A6B-9E78-4F26-9255-5C88636A0B04
                                   Attr: Legacy BIOS Bootable
     266240        4096       2  Label: "BIOS-BOOT"
                                   Type: BIOS Boot Partition
                                   UUID: EDA64B1B-A9BE-4F8C-89B0-55D2CC32845C
     270336     2097152       3  Label: "USR-A"
                                   Type: Alias for coreos-rootfs
                                   UUID: 7130C94A-213A-4E5A-8E26-6CCE9662F132
                                   Attr: priority=1 tries=0 successful=1
    2367488     2097152       4  Label: "USR-B"
                                   Type: Alias for coreos-rootfs
                                   UUID: E03DD35C-7C2D-4A47-B3FE-27F15780A57C
                                   Attr: priority=0 tries=0 successful=0
    4464640      262144       6  Label: "OEM"
                                   Type: Alias for linux-data
                                   UUID: E2096628-41E8-4ACB-A15E-45C1FF0D9132
    4726784      131072       7  Label: "OEM-CONFIG"
                                   Type: CoreOS reserved
                                   UUID: 2F78D549-025E-4412-B46A-A879EC644531
    4857856    11919327       9  Label: "ROOT"
                                   Type: CoreOS auto-resize
                                   UUID: 7D1B0DC8-00E2-40B6-8AFA-A15956FC2913
   16777183          32          Sec GPT table
   16777215           1          Sec GPT header
```

The following screenshot shows the `df -k` output in the same node:



```
core@ip-172-31-23-160 ~ $ df -k
Filesystem     1K-blocks    Used Available Use% Mounted on
devtmpfs          497168       0    497168   0% /dev
tmpfs             510100       0    510100   0% /dev/shm
tmpfs             510100     256    509844   1% /run
tmpfs             510100       0    510100   0% /sys/fs/cgroup
/dev/xvda9       5706380   20000   5410052   1% /
/dev/xvda3       1007760  354076    601668  38% /usr
tmpfs             510100       0    510100   0% /tmp
tmpfs             510100       0    510100   0% /media
/dev/xvda6        110576      60    101344   1% /usr/share/oem
```

The following are some notes on the preceding two outputs:

- There are nine partitions in total. The key partitions are `USR-A`, `USR-B`, `OEM`, and `ROOT`.
- System files are in the `USR` partition, user files are in the `ROOT` partition, and provider-related files are in the `OEM` partition.
- The `USR` partition is mounted as read-only, and the `ROOT` partition is mounted as read-write.
- The `ROOT` partition gets mounted as `/`, the `USR-A` or `USR-B` partition gets mounted in `/usr`, and the `OEM` partition gets mounted in `/usr/share/oem`.
- There are two `/usr` partitions, `USR-A` and `USR-B`. By default, the system comes up with the `USR-A` partition. When the CoreOS update is done, the root partition is downloaded to `USR-B` and, using persistent flags such as `priority`, `tries`, and `successful`, the CoreOS bootloader selects the appropriate USR partition on bootup. In the preceding example, the `USR-A` partition has priority set to `1` and the `USR-B` partition has priority set to `0`, and the CoreOS bootloader picks `USR-A`.

I did a manual update of OS and the following output shows the active partition being `USR-B` with priority for `USR-B` being higher. The manual update of the CoreOS system can be done using the command specified in the following *Update examples* section. The `/usr` directory is now pointing to `/dev/xvda4`, which is `USR-B`, and it was earlier pointing to `/dev/xvda3`, which was `USR-A`:

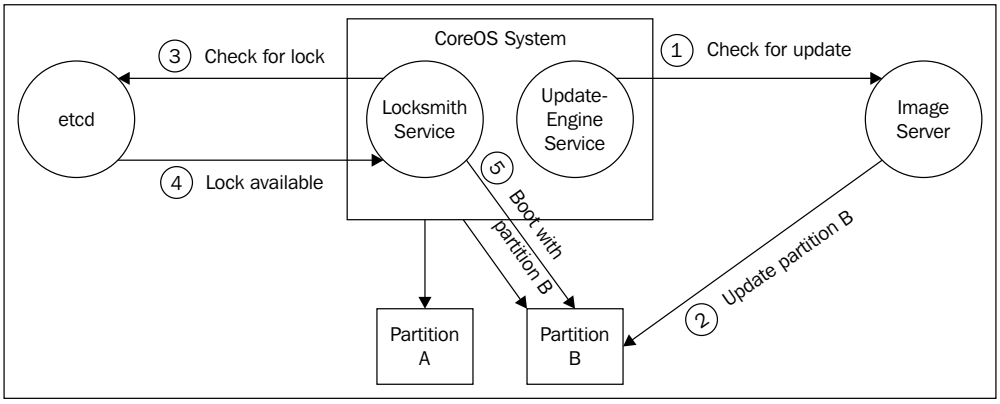
```
core@ip-172-31-39-190 ~ $ sudo cgpt show /dev/xvda
      start        size    part  contents
          0           1          Hybrid MBR
          1           1          Pri GPT header
          2          32          Pri GPT table
       4096      262144       1  Label: "EFI-SYSTEM"
                                   Type: EFI System Partition
                                   UUID: FACA3A6B-9E78-4F26-9255-5C88636A0B04
                                   Attr: Legacy BIOS Bootable
     266240        4096       2  Label: "BIOS-BOOT"
                                   Type: BIOS Boot Partition
                                   UUID: EDA64B1B-A9BE-4F8C-89B0-55D2CC32845C
     270336     2097152       3  Label: "USR-A"
                                   Type: Alias for coreos-rootfs
                                   UUID: 7130C94A-213A-4E5A-8E26-6CCE9662F132
                                   Attr: priority=1 tries=0 successful=1
    2367488     2097152       4  Label: "USR-B"
                                   Type: Alias for coreos-rootfs
                                   UUID: E03DD35C-7C2D-4A47-B3FE-27F15780A57C
                                   Attr: priority=2 tries=0 successful=0
    4464640      262144       6  Label: "OEM"
                                   Type: Alias for linux-data
                                   UUID: E2096628-41E8-4ACB-A15E-45C1FF0D9132
    4726784      131072       7  Label: "OEM-CONFIG"
                                   Type: CoreOS reserved
                                   UUID: 2F78D549-025E-4412-B46A-A879EC644531
    4857856    11919327       9  Label: "ROOT"
                                   Type: CoreOS auto-resize
                                   UUID: 7D1B0DC8-00E2-40B6-8AFA-A15956FC2913
   16777183          32          Sec GPT table
   16777215           1          Sec GPT header
core@ip-172-31-39-190 ~ $ df -k
Filesystem     1K-blocks   Used Available Use% Mounted on
devtmpfs          496552      0    496552   0% /dev
tmpfs             509796      0    509796   0% /dev/shm
tmpfs             509796    260    509536   1% /run
tmpfs             509796      0    509796   0% /sys/fs/cgroup
/dev/xvda9       5706380  20880   5409172   1% /
/dev/xvda4       1007760 398616    557128  42% /usr
/dev/xvda1        130798  58258     72540  45% /boot
tmpfs             509796      0    509796   0% /media
tmpfs             509796      0    509796   0% /tmp
/dev/xvda6        110576     60    101344   1% /usr/share/oem
```

CoreOS automatic update

CoreOS relies on the automatic update mechanism to keep the OS up to date. The following are some aspects of the CoreOS update:

- The CoreOS update mechanism is based on Google's open source Omaha protocol (`https://code.google.com/p/omaha/`) that is used in the Chrome browser.
- Either CoreOS public servers or private servers can be used as an image repository.
- The dual partition scheme is used where an update is done to the secondary partition while the primary partition is not touched. On reboot, there is a binary swap from the primary to the secondary partition. This keeps the update scheme robust. If there are issues with the new image, CoreOS automatically rolls back to the working image in the other partition.
- Images are signed and verified on each update.

The following screenshot shows you the steps for the automatic update:



Update and reboot services

There are two critical services controlling update and reboot in CoreOS. They are `update-engine.service` and `locksmithd.service`.

Update-engine.service

`Update-engine.service` takes care of periodically checking for updates from the appropriate release channel specified. A default check for update is done 10 minutes after reboot or at one-hour intervals.

The following output shows you the status of update-engine.service:

```
core@ip-172-31-23-160 ~ $ systemctl status update-engine.service
● update-engine.service - Update Engine
   Loaded: loaded (/usr/lib64/systemd/system/update-engine.service; disabled; vendor preset: disabled)
   Active: active (running) since Sat 2015-09-19 06:46:19 UTC; 13min ago
 Main PID: 477 (update_engine)
   Memory: 7.3M
      CPU: 88ms
   CGroup: /system.slice/update-engine.service
           └─477 /usr/sbin/update_engine -foreground -logtostderr
```

The release channel is specified in `/etc/coreos/update.conf`. In the following node, the release channel is selected as stable. The release channel is derived from `cloud-config`:

```
core@ip-172-31-23-160 ~ $ cat /etc/coreos/update.conf
GROUP=stable
```

`Update-engine.service` takes care of updating the appropriate partition, `USR-A` or `USR-B`. The currently used partition is not touched.

The following command can be executed to trigger a manual update:

```
update_engine_client -check_for_update
```

Debugging update-engine.service

Logs for the update service can be checked using `journalctl -u update-engine.service`. From the logs, we can identify the Omaha protocol request and response, and debugging can be done using error codes in the response.

Locksmithd.service

`Locksmithd.service` takes care of rebooting the CoreOS node using the selected reboot strategy. `Locksmithd.service` runs as a daemon.

The following output shows you the status of `locksmithd.service`:

```
core@ip-172-31-23-160 ~ $ systemctl status locksmithd.service
● locksmithd.service - Cluster reboot manager
   Loaded: loaded (/usr/lib64/systemd/system/locksmithd.service; disabled; vendor preset: disabled)
   Active: active (running) since Sat 2015-09-19 06:46:20 UTC; 29min ago
 Main PID: 507 (locksmithd)
   Memory: 6.2M (limit: 32.0M)
      CPU: 18ms
   CGroup: /system.slice/locksmithd.service
           └─507 /usr/lib/locksmith/locksmithd
```

Locksmith strategy

The following are the four configurable strategies for the CoreOS node reboot after a new image update.

The etcd-lock scheme

In this scheme, the reboot is done after first taking a lock from `etcd`. In a multinode cluster, this works out really well as it prevents all the nodes from rebooting at the same time and maintains cluster integrity. We can control the number of nodes that can reboot together using the lock count mechanism. The lock `max` count specifies the number of nodes that can acquire a lock simultaneously. In a three-node cluster, we need to limit the lock `max` count to `1`, but in a five-node cluster, we can keep the lock `max` count up to `2`, which allows a maximum of two nodes to acquire lock and reboot simultaneously.

The following example shows you how the available lock count varies when we do the locking and unlocking operation:

```
core@ip-172-31-23-160 /etc/locksmithd $ locksmithctl status
Available: 1
Max: 1
core@ip-172-31-23-160 /etc/locksmithd $ locksmithctl lock
core@ip-172-31-23-160 /etc/locksmithd $ locksmithctl status
Available: 0
Max: 1

MACHINE ID
ea3dc076d4784e64ad09c928825c0810
core@ip-172-31-23-160 /etc/locksmithd $ locksmithctl unlock
core@ip-172-31-23-160 /etc/locksmithd $ locksmithctl status
Available: 1
Max: 1
```

Reboot

In this scheme, the node is rebooted immediately without taking a lock from the cluster. This is useful in scenarios where the upgrade is manually controlled by the administrator.

Best-effort

In this scheme, it is first checked whether `etcd` is running. If `etcd` is running, the `etcd` lock is acquired and then the rebooting is done. Otherwise, reboot is done immediately. This is a variation of the etcd-lock scheme mentioned before.

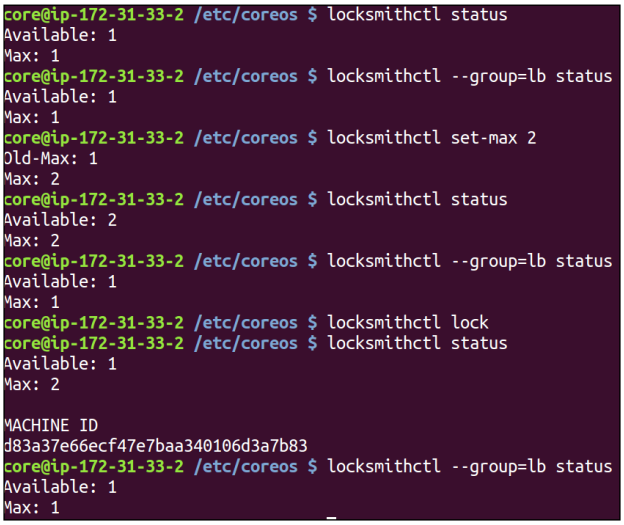
Off

This causes locksmithd to exit and do nothing. This option should not be chosen unless the administrator wants to control the upgrades with great precision.

Groups

Locksmith groups were introduced in locksmithd version 0.3.1. With groups, we can group a set of CoreOS nodes and locking will be applicable to this group. For example, let's say that we have a five-node cluster and two nodes are running load balancers. If we set the lock-max-count to `2`, it is possible that both the nodes running load balancers can reboot at the same time and we can lose that service during this period. To avoid this issue, we can set a different lock `max` count for the default group and the `lb` group.

In the example shown in following screenshot, we have set the lock count of `2` for the default group and lock count of `1` for the `lb` group. Groups can be defined as part of starting the `locksmithd service`. To put a CoreOS node in a locksmith group, we need to start locksmithd with the `--group` option or set the `LOCKSMITHD_GROUP` environment variable and restart the locksmithd service:



Locksmithctl

Locksmithctl is a frontend CLI to control locksmith. Using this, we can get the status of locksmith service, lock and unlock groups, set the `lock max` count, and so on.

Debugging locksmithd.service

Logs for this service can be checked with `journalctl -u locksmithd.service`.

Setting update options

CoreOS update options can be set using either `cloud-config` or by changing configuration files manually. Using `cloud-config`, update options are configured as part of the node configuration after reboot. With the manual approach, we need to start the appropriate update services for changes to take effect. The manual approach is used mainly to debug.

Using cloud-config

The following is a sample `cloud-config` with the release channel group set to `stable` and the locksmith reboot strategy set to `etcd-lock`. (The default server used is `https://public.update.core-os.net/`, so this is not specified here.)

```
#cloud-config
coreos:
  etcd2:
    # specify the initial size of your cluster with ?size=X
    discovery: https://discovery.etcd.io/
eb32a1397bd087f84e65ab802b6aa2f7
    advertise-client-urls: http://$private_ipv4:2379,http://$private_
ipv4:4001
    initial-advertise-peer-urls: http://$private_ipv4:2380
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  update:
    reboot-strategy: "etcd-lock"
    group: "stable"
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

After starting the cluster using the preceding `cloud-config`, we can check whether `/etc/coreos/update.conf` is updated with the correct parameters:

```
core@ip-172-31-23-162 ~ $ cat /etc/coreos/update.conf
GROUP=stable
REBOOT_STRATEGY=etcd-lock
core@ip-172-31-23-162 ~ $ sudo systemctl restart locksmithd.service
```

Manual configuration

The default reboot strategy is `best-effort`. In the following node, the reboot strategy is not specified, so it is using best-effort:

```
core@ip-172-31-23-162 ~ $ cat /etc/coreos/update.conf
GROUP=stable
```

Let's change the reboot strategy to `reboot` in `/etc/coreos/update.conf`. We need to restart `locksmithd.service`:

```
core@ip-172-31-34-115 /etc/coreos $ cat update.conf
GROUP=stable
REBOOT_STRATEGY=reboot
core@ip-172-31-34-115 /etc/coreos $ sudo systemctl restart locksmithd.service
```

As shown in the following logs, the reboot strategy is taking effect:

```
core@ip-172-31-34-115 /etc/coreos $ journalctl -u locksmithd.service -n 5 --no-pager
-- Logs begin at Sat 2015-09-19 10:15:38 UTC, end at Sat 2015-09-19 10:47:30 UTC. --
Sep 19 10:46:50 ip-172-31-34-115.us-west-2.compute.internal systemd[1]: Stopping Cluster reboot manager...
Sep 19 10:46:50 ip-172-31-34-115.us-west-2.compute.internal locksmithd[580]: Received interrupt/termination signal - shutting down.
Sep 19 10:46:50 ip-172-31-34-115.us-west-2.compute.internal systemd[1]: Started Cluster reboot manager.
Sep 19 10:46:50 ip-172-31-34-115.us-west-2.compute.internal systemd[1]: Starting Cluster reboot manager...
Sep 19 10:46:50 ip-172-31-34-115.us-west-2.compute.internal locksmithd[925]: locksmithd starting currentOperation="UPDATE_STATUS_IDLE" strategy="reboot"
```

Update examples

We can do updates within the same release channel or across release channels. If we do updates in the same release channel, the node gets updated to the latest version in that release channel. If we do updates across release channels, the node gets updated to the latest version in the new release channel.

Updating within the same release channel

Let's look at the initial version and reboot strategy. The node is running stable version 723.3.0 as shown in the following screenshot:

```
core@ip-172-31-34-117 ~ $ cat /etc/os-release
NAME=CoreOS
ID=coreos
VERSION=723.3.0
VERSION_ID=723.3.0
BUILD_ID=
PRETTY_NAME="CoreOS 723.3.0"
ANSI_COLOR="1;32"
HOME_URL="https://coreos.com/"
BUG_REPORT_URL="https://github.com/coreos/bugs/issues"
core@ip-172-31-34-117 ~ $ cat /etc/coreos/update.conf
GROUP=stable
REBOOT_STRATEGY=etcd-lock _
```

Looking at the CoreOS releases page, the latest STABLE release is 766.3.0. If we do an update on the STABLE channel, the node should get updated to 766.3.0.

Let's trigger the update manually with the following command:

```
update_engine_client -check_for_update
```

If we don't trigger the update manually, `update-engine` will still do the update based on its periodic checks.

The following logs from `update-engine.service` show the Omaha request to the CoreOS public imaging server:

```
Sep 20 03:47:48 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: [0920/034748:INFO:update_check_scheduler.cc(82)] Next update check in 49m17s
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: [0920/043236:INFO:dbus_service.cc(57)] Attempting interactive update
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: [0920/043236:INFO:update_attempter.cc(256)] New update check requested
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: [0920/043236:INFO:omaha_request_params.cc(60)] Current group set to stable
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: [0920/043236:INFO:update_attempter.cc(475)] Already updated boot flags. Skipping.
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: [0920/043236:INFO:update_attempter.cc(658)] Scheduling an action processor start.
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: [0920/043236:INFO:action_processor.cc(36)] ActionProcessor::StartProcessing: OmahaRequestAction
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: [0920/043236:INFO:omaha_request_action.cc(257)] Posting an Omaha request to https://public.update.cor
e-os.net/v1/update/
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: [0920/043236:INFO:omaha_request_action.cc(258)] Request: <?xml version="1.0" encoding="UTF-8"?>
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: <request protocol="3.0" version="CoreOSUpdateEngine-0.1.0.0" updaterversion="CoreOSUpdateEngine-0.1.0
.0" installsource="ondemandupdate" ismachine="1">
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: <os version="Chateau" platform="CoreOS" sp="723.3.0_x86_64"></os>
```

The following logs from update-engine show the Omaha response from the CoreOS public server giving the image with version 766.3.0:

```
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: <response protocol="3.0" server="update.core-os.net">
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: <daystart elapsed_seconds="0"></daystart>
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: <app appid="e96281a6-d1af-4bde-9a0a-97b76e56dc57" status="ok">
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: <updatecheck status="ok">
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: <urls>
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: <url codebase="https://commondatastorage.googleapis.com/update-storage.core-os.net/amd64-usr/766.3.0/
"></url>
Sep 20 04:32:36 ip-172-31-34-117.us-west-2.compute.internal update_engine[580]: </urls>
```

After the update is successful, the following message appears on the node from `locksmithd.service`, indicating that the node will be updated with the new image. The new image is updated to the non-active USR partition:

```
Broadcast message from locksmithd at 2015-09-20 04:40:41.819947382 +0000 UTC:
System reboot in 5 minutes!
```

The following is the node version after reboot. We can see that the version is upgraded successfully to 766.3.0:

```
core@ip-172-31-34-117 ~ $ cat /etc/os-release
NAME=CoreOS
ID=coreos
VERSION=766.3.0
VERSION_ID=766.3.0
BUILD_ID=
PRETTY_NAME="CoreOS 766.3.0"
ANSI_COLOR="1;32"
HOME_URL="https://coreos.com/"
BUG_REPORT_URL="https://github.com/coreos/bugs/issues"
```

Updating from one release channel to another

We can switch release channels by updating `/etc/coreos/update.conf`. These are the steps:

- Update the release channel group from `stable` to `alpha`, as shown in the following screenshots:

```
core@ip-172-31-34-117 ~ $ cat /etc/coreos/update.conf
GROUP=alpha
REBOOT_STRATEGY=etcd-lock _
```

- Restart `update-engine.service`:

```
sudo systemctl restart update-engine
```

- The `update-engine` service will check for an update after 10 minutes. We can force the update with the following command:

```
update_engine_client -check_for_update
```

The following log shows you that the version 808.0.0 image is being fetched now:

```
Sep 20 04:53:04 ip-172-31-34-117.us-west-2.compute.internal update_engine[838]: <response protocol="3.0" server="update.core-os.net">
Sep 20 04:53:04 ip-172-31-34-117.us-west-2.compute.internal update_engine[838]: <daystart elapsed_seconds="0"></daystart>
Sep 20 04:53:04 ip-172-31-34-117.us-west-2.compute.internal update_engine[838]: <app appid="e96281a6-d1af-4bde-9a0a-97b76e56dc57" status="ok">
Sep 20 04:53:04 ip-172-31-34-117.us-west-2.compute.internal update_engine[838]: <updatecheck status="ok">
Sep 20 04:53:04 ip-172-31-34-117.us-west-2.compute.internal update_engine[838]: <urls>
Sep 20 04:53:04 ip-172-31-34-117.us-west-2.compute.internal update_engine[838]: <url codebase="https://commondatastorage.googleapis.com/update-storage.core-os.net/amd64-usr/808.0.0/"></url>
Sep 20 04:53:04 ip-172-31-34-117.us-west-2.compute.internal update_engine[838]: </urls>
Sep 20 04:53:04 ip-172-31-34-117.us-west-2.compute.internal update_engine[838]: <manifest version="808.0.0">
```

The following is the version after the node is rebooted. We can see that the image is upgraded to the latest alpha release `808.0.0`:

```
core@ip-172-31-34-117 ~ $ cat /etc/os-release
NAME=CoreOS
ID=coreos
VERSION=808.0.0
VERSION_ID=808.0.0
BUILD_ID=
PRETTY_NAME="CoreOS 808.0.0"
ANSI_COLOR="1;32"
HOME_URL="https://coreos.com/"
BUG_REPORT_URL="https://github.com/coreos/bugs/issues"
```

CoreUpdate

CoreUpdate is a commercial service provided by CoreOS to manage the customer updates of CoreOS clusters. The following are some of the features provided by the CoreUpdate service:

- The GUI dashboard provides you with a summary and detailed view of all the updates.
- Custom image servers will be provided on a per customer basis.
- Server groups can be created so that updates can be done in groups and rate limiting can be done on a per group basis.
- An HTTP API is provided so that CoreUpdate can be integrated with existing DevOps systems available with the customer.
- Images can be hosted on public servers or customer's local servers. This is useful from a security perspective so that customers don't have to worry about opening up their firewall.
- `Updateservicectl` is provided as a frontend CLI.

Vagrant CoreOS update

If the Vagrant box is already downloaded, the new CoreOS version will be updated only if the box is updated.

Even though we change the version in Vagrantfile from stable to alpha to beta, the new CoreOS version does not get updated on `vagrant reload --provision`. Only when we perform vagrant destroy and restart, the new version gets loaded. We can directly trigger an update from the CoreOS node using `update-engine`, and it works irrespective of the VBOX version.

We get the following message when Vagrant CoreOS is not up to date:

```
==> core-01: Checking if box 'coreos-stable' is up to date...
==> core-01: A newer version of the box 'coreos-stable' is available! You currently
==> core-01: have version '723.3.0'. The latest is version '766.3.0'. Run
==> core-01: `vagrant box update` to update.
```

To update the Vagrant box version, we can perform `vagrant box update` as shown in the following screenshot:

```
$ vagrant box update
==> core-01: Checking for updates to 'coreos-stable'
    core-01: Latest installed version: 723.3.0
    core-01: Version constraints:
    core-01: Provider: virtualbox
==> core-01: Updating 'coreos-stable' with provider 'virtualbox' from version
==> core-01: '723.3.0' to '766.3.0'...
```

The `vagrant reload` command or `vagrant reload --provision` command do not help to update the CoreOS version. We need to destroy and recreate the cluster to get the latest version.

Summary

In this chapter, we covered different aspects of the CoreOS update, including the CoreOS release cycle, services controlling the CoreOS update, and options available to customers to control their cluster's update strategy. The CoreOS update mechanism is simple, unique, and robust, and it takes care of the biggest concern in the cloud, which is security. In the next chapter, we will cover details on critical CoreOS services—`systemd`, `etcd`, and `fleet`.

References

- CoreOS releases: `https://coreos.com/releases/`
- CoreOS update philosophy: `https://coreos.com/using-coreos/updates/`
- CoreUpdate service: `https://coreos.com/products/coreupdate/`
- Locksmith GitHub page: `https://github.com/coreos/locksmith`
- Update strategies: `https://coreos.com/os/docs/latest/update-strategies.html`

Further reading and tutorials

- The anatomy of a CoreOS update: `https://www.youtube.com/watch?v=JeICd9XyXfY`
- The Omaha update protocol: `https://github.com/google/omaha` and `https://coreos.com/docs/coreupdate/custom-apps/coreupdate-protocol/`

4

CoreOS Primary Services – Etcd, Systemd, and Fleet

This chapter will cover the internals of CoreOS' critical services—Etcd, Systemd, and Fleet. For each of the services, we will cover installation, configuration, and their applications. CoreOS ships with Etcd, Systemd, and Fleet by default. They can also be installed as standalone components in any Linux system. The following topics will be covered in this chapter:

- Etcd—installation, access methods, configuration options, use cases, tuning, cluster management, security, authentication, and debugging
- Systemd—unit types, specifiers, templates, and special units
- Fleet—installation, access methods, templates, scheduling, HA, and debugging
- Service discovery options using Etcd and Fleet

Etcd

Etcd is a distributed key-value store used by all the machines in the CoreOS cluster to read/write and exchange data. An overview of etcd is provided in *Chapter 1*, *CoreOS Overview*. This section will cover the internals of etcd.

Versions

Etcd is under continuous development, and frequent releases are done to add enhancements as well as fix bugs. The following are some major updates from recent etcd releases:

- Version 2.0 is the first stable release and was released in January 2015. Pre-version 2.0 is available as etcd and post-version 2.0 is available as etcd2 in CoreOS nodes.
- Version 2.0 added IANA-assigned ports `2379` for client-to-server communication and `2380` for server-to-server communication. Previously, port `4001` was used for client-to-server communication and port `7001` was used for server-to-server communication.
- Version 2.1 introduced authentication and metrics collection features and these are in experimental mode.
- The latest release as of September 2015 is 2.2.0.
- An experimental v3 API (some examples are multikey reads, range reads, and binary keys) is available now as a preview and will be available officially in version 2.3.0 scheduled at the end of October 2015.

All examples in this chapter are based on etcd version 2.1.0 and above.

Installation

CoreOS ships with etcd. Both the etcd and etcd2 versions are available in the base CoreOS image. The following are the etcd versions available in the CoreOS alpha image 779.0.0:

```
core@core-01 ~ $ cat /etc/os-release
NAME=CoreOS
ID=coreos
VERSION=779.0.0
VERSION_ID=779.0.0
BUILD_ID=
PRETTY_NAME="CoreOS 779.0.0"
ANSI_COLOR="1;32"
HOME_URL="https://coreos.com/"
BUG_REPORT_URL="https://github.com/coreos/bugs/issues"
core@core-01 ~ $ cat /etc/coreos/update.conf
GROUP=alpha
core@core-01 ~ $ etcd --version
etcd version 0.4.9
core@core-01 ~ $ etcd2 --version
etcd Version: 2.1.1
Git SHA: 6335fdc
Go Version: go1.4.2
Go OS/Arch: linux/amd64
```

Standalone installation

Etcd can also be installed on any Linux machine. The following is the installation command tried out on Ubuntu 14.04 to install etcd version 2.2:

```
curl -L  https://github.com/coreos/etcd/releases/download/v2.2.0/etcd-v2.2.0-linux-amd64.tar.gz -o etcd-v2.2.0-linux-amd64.tar.gz
tar xzvf etcd-v2.2.0-linux-amd64.tar.gz
```

The following example shows you how to try out etcd in the standalone mode. To start the server run the following command:

```
etcd -name etcdtest
```

Now, check whether we can connect to the etcd server using some basic commands:

```
etcdctl cluster-health
```

The following screenshot is the output of the preceding command:

```
smakam14@jungle1:~$ etcdctl cluster-health
cluster is healthy
member ce2a822cea30bfca is healthy
smakam14@jungle1:~$ etcdctl member list
ce2a822cea30bfca: name=etcdtest peerURLs=http://localhost:2380,http://localhost:7001 clientURLs=http://localhost:2379,http://localhost:4001
```

The following is an example of a simple set and get operation using the curl interface:

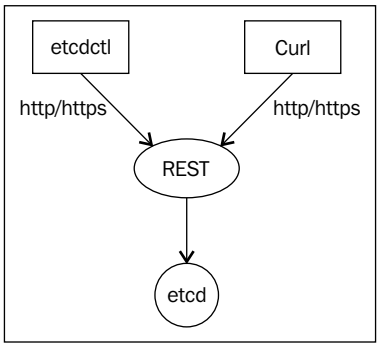
```
curl -L -X PUT http://127.0.0.1:2379/v2/keys/message -d value="hello"
curl -L http://127.0.0.1:2379/v2/keys/message
```

The following screenshot is the output of the preceding command:

```
smakam14@jungle1:~$ curl -L -X PUT http://127.0.0.1:2379/v2/keys/message -d value="hello"
{"action":"set","node":{"key":"/message","value":"hello","modifiedIndex":4,"createdIndex":4}}
smakam14@jungle1:~$ curl -L http://127.0.0.1:2379/v2/keys/message
{"action":"get","node":{"key":"/message","value":"hello","modifiedIndex":4,"createdIndex":4}}
```

Accessing etcd

Etcd can be accessed using either etcdctl CLI or REST API. This applies to both the standalone etcd as well as etcd in CoreOS. The following figure shows you the different ways to access etcd:



REST

The etcd database can be accessed and modified through the REST API. The etcd database can be accessed either locally or remotely using this approach.

The following example shows the `curl` method to access the CoreOS node to get all the keys:

```
curl -L http://localhost:2379/v2/keys/?recursive=true
```

The following screenshot is the output of the preceding command:

```
core@core-01 ~ $ curl -L http://localhost:2379/v2/keys/?recursive=true
{"action":"get","node":{"dir":true,"nodes":[{"key":"/coreos.com","dir":true,"nodes":[{"key":"/coreos.com
/updateengine","dir":true,"nodes":[{"key":"/coreos.com/updateengine/rebootlock","dir":true,"nodes":[{"ke
y":"/coreos.com/updateengine/rebootlock/semaphore","value":"{\"semaphore\":1,\"max\":1,\"holders\":[]}",
"modifiedIndex":3964,"createdIndex":5}],"modifiedIndex":5,"createdIndex":5}],"modifiedIndex":5,"createdI
ndex":5}],"modifiedIndex":5,"createdIndex":5}]}}
```

The following example shows the `curl` method to access the remote CoreOS node to get all the keys:

```
curl -L http://172.17.8.101:2379/v2/keys/?recursive=true
```

The following screenshot is the output of the preceding command:

```
smakam14@jungle1:~$ curl -L http://172.17.8.101:2379/v2/keys/?recursive=true
{"action":"get","node":{"dir":true,"nodes":[{"key":"/coreos.com","dir":true,"nodes":[{"key":"/coreos.com/updateengine","dir":true,"nodes":[{"key":
"/coreos.com/updateengine/rebootlock","dir":true,"nodes":[{"key":"/coreos.com/updateengine/rebootlock/semaphore","value":"{\"semaphore\":1,\"max\"
:1,\"holders\":[]}","modifiedIndex":3964,"createdIndex":5}],"modifiedIndex":5,"createdIndex":5}],"modifiedIndex":5,"createdIndex":5}],"modifiedInd
ex":5,"createdIndex":5}]}}
```

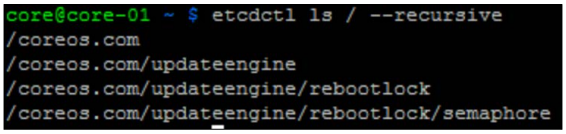
Etcdctl

Etcdctl is a CLI wrapper on top of the REST interface. Etcdctl can be used for local or remote access.

The following example shows etcdctl method to access the CoreOS node to get all the keys:

```
etcdctl ls / --recursive
```

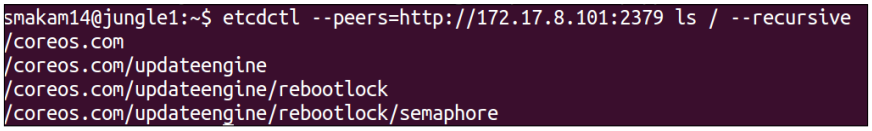
The following screenshot is the output of the preceding command:



The following example shows etcdctl method to access the remote CoreOS node to get all the keys:

```
etcdctl --peers=http://172.17.8.101:2379 ls / --recursive
```

The following screenshot is the output of the preceding command:



Etcd configuration

Etcd configuration parameters can be used to modify the etcd member property or cluster-wide property. Etcd options can be set either in the command line or using environment variables. The command line will override the environment variables. The following are the broad categories and their critical configuration parameters/ environment variables:

- Member: Name, data-dir, and heartbeat interval
- Cluster: Discovery token and initial cluster nodes
- Proxy: Proxy on/off and proxy intervals
- Security: Certificate and key
- Logging: Enable/disable logging and logging levels
- Experimental

The following is an etcd invocation example, where we use some of the preceding configuration parameters:

```
etcd -name infra0 -data-dir infra0  --cacert=~/.etcd-ca/ca.crt -cert-
file=/home/smakam14/infra0.crt -key-file=/home/smakam14/infra0.key.
insecure  -advertise-client-urls=https://192.168.56.104:2379 -listen-
client-urls=https://192.168.56.104:2379
```

Etcd environment variables can also be specified in `cloud-config`. The following is a `cloud-config` example to specify etcd environment variables:

```
etcd2:
    #generate a new token for each unique cluster from
https://discovery.etcd.io/new
    discovery: https://discovery.etcd.io/
d93c8c02eedadddd3cf14828f9bec01c
    # multi-region and multi-cloud deployments need to use $public_
ipv4
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    # listen on both the official ports and the legacy ports
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
```

Etcd2 environment variables from cloud-config are stored in the following directory: `/run/systemd/system/etcd2.service.d`.

The etcd2 service needs to be restarted if the environment variables are changed.

A complete list of configuration parameters and environment variables for etcd can be found at `https://coreos.com/etcd/docs/latest/configuration.html`.

Etcd operations

The following are some examples of major operations that can be done using etcd:

- Set, get, and delete operations of a key-value pair
- Set a key with timeout where the key expires automatically
- Set a key based on the atomic condition check
- Hidden keys
- Watching and waiting for key changes
- Creating in-order keys

Using these operations, etcd can be used for a variety of distributed application use cases. The following is an example TTL use case where we check for the liveliness of the Apache service and update service details such as the IP address and port number in etcd, which other applications can use to determine if the service is running or not. If the Apache service dies, the etcd key-value pair will be deleted after 30 seconds in this case:

```
## Test whether service is accessible and then register useful
information like IP address, port
ExecStart=/bin/bash -c '\
  while true; do \
    curl -f ${COREOS_PUBLIC_IPV4}:%i; \
    if [ $? -eq 0 ]; then \
      etcdctl set /services/apachet/${COREOS_PUBLIC_IPV4} \'{"host":
"%H", "ipv4_addr": ${COREOS_PUBLIC_IPV4}, "port": %i}\' --ttl 30; \
    fi; \
    sleep 20; \
  done'
```

We can find statistics about the etcd node as well as the key-related operations.

The following output shows the etcd node statistics:

```
curl http://127.0.0.1:2379/v2/stats/self | jq .
```

The following screenshot is the output of the preceding command:

```
core@core-02 ~ $ curl http://127.0.0.1:2379/v2/stats/self | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   378  100   378    0     0  34832      0 --:--:-- --:--:-- --:--:-- 37800
{
  "name": "6c6054e6e10440ed86038e61d739f77b",
  "id": "eb553ce6ac28c43f",
  "state": "StateLeader",
  "startTime": "2015-09-26T05:01:53.788912384Z",
  "leaderInfo": {
    "leader": "eb553ce6ac28c43f",
    "uptime": "1h52m25.941974851s",
    "startTime": "2015-09-26T05:02:26.615173693Z"
  },
  "recvAppendRequestCnt": 0,
  "sendAppendRequestCnt": 100036,
  "sendPkgRate": 9.760212714708366,
  "sendBandwidthRate": 181082.25131831938
}
```

The following output shows the etcd key statistics:

```
curl http://127.0.0.1:2379/v2/stats/store | jq .
```

The following screenshot is the output of the preceding command:

```
core@core-02 ~ $ curl http://127.0.0.1:2379/v2/stats/store | jq '.'
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100   309  100   309    0     0  41610      0 --:--:-- --:--:-- --:--:-- 44142
{
  "getsSuccess": 18682,
  "getsFail": 60723,
  "setsSuccess": 30,
  "setsFail": 0,
  "deleteSuccess": 19,
  "deleteFail": 0,
  "updateSuccess": 5062,
  "updateFail": 24,
  "createSuccess": 42,
  "createFail": 37,
  "compareAndSwapSuccess": 13843,
  "compareAndSwapFail": 3,
  "compareAndDeleteSuccess": 3,
  "compareAndDeleteFail": 1,
  "expireCount": 9,
  "watchers": 2
}
```

Etcd tuning

The following are some etcd parameters that can be tuned to achieve optimum cluster performance based on the operating environment:

- **Cluster size**: A bigger cluster size provides you with better redundancy. The disadvantage with big cluster sizes is that updates can take a long time. In *Chapter 1*, *CoreOS Overview*, we saw the failure tolerance limit with different cluster sizes.
- **Heartbeat interval**: This is the time interval at which the master node sends a heartbeat message to its followers. The default heartbeat interval is 100 ms. It is necessary to choose a heartbeat interval based on the average round-trip time taken for the ping between nodes. If the nodes are geographically distributed, then the round-trip time will be longer. The suggested heartbeat interval is 0.5-1.5 x the average round-trip time. If we choose a small heartbeat interval, the overhead will be a higher number of packets. If we choose a large heartbeat interval, it will take a longer time to detect leader failure. The heartbeat interval can be set using the `heartbeat-interval` parameter in the etcd command line or the `ETCD_HEARTBEAT_INTERVAL` environment variable.

- **Election timeout**: When the follower nodes fail to get a heartbeat message for the election timeout value, they become the leader node. The default election timeout is 1,000 ms. The suggested value for election timeout is 10 times the heartbeat interval. Keeping the election timeout too low can cause false leader election. The election timeout can be set using the `election-timeout` parameter in the etcd command line or the `ETCD_ELECTION_TIMEOUT` environment variable.

Etcd proxy

An etcd proxy is used when worker nodes want to use the master node or master cluster to provide etcd service. In this case, all etcd requests from the worker node are proxied to the master node and the master node replies to the worker node.

Let's say that we have a working three-node master cluster as follows:

```
core@core-01 ~ $ etcdctl member list
ef5fbdd70d2977: name=0a697ef95af44929bfaa315fbcec8211 peerURLs=http://172.17.8.103:2380 clientURLs=http:
//172.17.8.103:2379
63f6909788785a99: name=0622ba3706f8417d90faec9970a7f066 peerURLs=http://172.17.8.101:2380 clientURLs=htt
p://172.17.8.101:2379
eb553ce6ac28c43f: name=6c6054e6e10440ed86038e61d739f77b peerURLs=http://172.17.8.102:2380 clientURLs=htt
p://172.17.8.102:2379
```

The following example shows the `cloud-config` for the fourth node that is a worker node and acting as a proxy. Here, the master cluster members are mentioned statically:

```
#cloud-config
coreos:
  etcd2:
    proxy: on
    listen-client-urls: http://localhost:2379
    initial-cluster: etcdserver=http://172.17.8.101:2380,
http://172.17.8.102:2380, http://172.17.8.103:2380
  fleet:
    etcd_servers: "http://localhost:2379"
    public-ip: $public_ipv4
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

In the preceding *Etcd configuration* section, we have turned on the proxy and pointed to the `etcd_server` cluster. The fourth node needs to be started with the preceding `cloud-config`.

The following example shows the `cloud-config` for the fourth node that is acting as a proxy and using a discovery token. We need to use the same discovery token as we did for the three-node cluster:

```
#cloud-config
coreos:
  etcd2:
    proxy: on
    # use the same discovery token as for master, these nodes will
proxy to master
    discovery: <your token>
    # listen on both the official ports and the legacy ports
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
  fleet:
    etcd_servers: "http://localhost:2379"
    public-ip: $public_ipv4
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

The following is the etcd member output in the new node. As we can see, the etcd cluster is composed of only three nodes and the new node is proxying to the master etcd cluster:

```
core@corew-01 ~ $ etcdctl member list
ef5fbdd70d2977: name=0a697ef95af44929bfaa315fbcec8211 peerURLs=http://172.17.8.103:2380 clientURLs=http:
//172.17.8.103:2379
63f6909788785a99: name=0622ba3706f8417d90faec9970a7f066 peerURLs=http://172.17.8.101:2380 clientURLs=htt
p://172.17.8.101:2379
eb553ce6ac28c43f: name=6c6054e6e10440ed86038e61d739f77b peerURLs=http://172.17.8.102:2380 clientURLs=htt
p://172.17.8.102:2379
```

The following is the Fleet machine's output in the new node. As we can see, there are four nodes and this includes the fourth worker node and the three-node etcd cluster:

```
core@corew-01 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
0622ba37...     172.17.8.101    -
0a697ef9...     172.17.8.103    -
6c6054e6...     172.17.8.102    -
77438148...     172.17.8.106    -
```

Adding and removing nodes from a cluster

There will be scenarios where we need to add and remove nodes from a working etcd cluster. This section illustrates how to add and remove nodes in a working etcd cluster.

Let's say that we have a three-node working cluster and we want to add a fourth node to the cluster. The following command can be executed in one of the three working nodes to add the fourth node detail:

```
core@core-01 ~ $ etcdctl member add core-04 http://172.17.8.104:2380
Added member named core-04 with ID a082caf74f222324 to cluster

ETCD_NAME="core-04"
ETCD_INITIAL_CLUSTER="0a697ef95af44929bfaa315fbcec8211=http://172.17.8.103:2380,0622ba3706f8417d90faec9
70a7f066=http://172.17.8.101:2380,core-04=http://172.17.8.104:2380,6c6054e6e10440ed86038e61d739f77b=htt
://172.17.8.102:2380"
ETCD_INITIAL_CLUSTER_STATE="existing"
```

The following `cloud-config` can be used to start the new fourth node:

```
#cloud-config
coreos:
  etcd2:
    name: core-04
    initial_cluster: "core-01=http://172.17.8.101:2380,core-
02=http://172.17.8.102:2380,core-03=http://172.17.8.103:2380,core-
04=http://172.17.8.104:2380"
    initial_cluster_state: existing
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    # listen on both the official ports and the legacy ports
    # legacy ports can be omitted if your application doesn't depend
on them
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  fleet:
    public-ip: $public_ipv4
  units:
    # Note: this requires a release that contains etcd2
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
```

From the following output, we can see that the new member has been successfully added:

```
core@core-04 ~ $ etcdctl cluster-health
member ef5fbdd70d2977 is healthy: got healthy result from http://172.17.8.103:2379
member 63f6909788785a99 is healthy: got healthy result from http://172.17.8.101:2379
member a082caf74f222324 is healthy: got healthy result from http://172.17.8.104:2379
member eb553ce6ac28c43f is healthy: got healthy result from http://172.17.8.102:2379
cluster is healthy
```

The following command can be used to remove the fourth number that we added before:

```
core@core-01 ~ $ etcdctl member remove a082caf74f222324
Removed member a082caf74f222324 from cluster
```

Let's check the member list and cluster health now. We can see that the three nodes are part of the cluster and that the fourth node has been removed:

```
core@core-01 ~ $ etcdctl member list
ef5fbdd70d2977: name=0a697ef95af44929bfaa315fbcec8211 peerURLs=http://172.17.8.103:2380 clientURLs=http:
//172.17.8.103:2379
63f6909788785a99: name=0622ba3706f8417d90faec9970a7f066 peerURLs=http://172.17.8.101:2380 clientURLs=htt
p://172.17.8.101:2379
eb553ce6ac28c43f: name=6c6054e6e10440ed86038e61d739f77b peerURLs=http://172.17.8.102:2380 clientURLs=htt
p://172.17.8.102:2379
core@core-01 ~ $ etcdctl cluster-health
member ef5fbdd70d2977 is healthy: got healthy result from http://172.17.8.103:2379
member 63f6909788785a99 is healthy: got healthy result from http://172.17.8.101:2379
member eb553ce6ac28c43f is healthy: got healthy result from http://172.17.8.102:2379
cluster is healthy
```

Node migration and backup

Node migration is necessary to handle failure of the node and cluster and also to replicate the cluster to a different location.

To take a backup of the etcd database, we can perform the following:

```
Sudo etcdctl backup --data-dir=/var/lib/etcd2 --backup-dir=/tmp/etcd2
```

This approach allows us to reuse the backed-up etcd data in another cluster. In this approach, `nodeid` and `clusterid` are overwritten in the backup directory to prevent unintentional addition of a new node to the old cluster.

To preserve the node ID and cluster ID, we have to manually make a copy, and the copy can be used to restart the service.

The following are the steps to move the etcd2 data directory:

1. Stop the service:

```
core@core-02 ~ $ sudo systemctl stop etcd2.service
```

2. Make a copy of the `/var/lib/etcd2` etcd data directory in `/tmp/etcd2_backup`:

```
core@core-02 ~ $ sudo mkdir /tmp/etcd2_backup
core@core-02 ~ $ sudo cp -r /var/lib/etcd2/* /tmp/etcd2_backup/
```

3. Start etcd2 manually using the new data directory, `/tmp/etcd2_backup`:

```
core@core-02 ~ $ sudo etcd2 --discovery 'https://discovery.etcd.io/99ce5d499df6a4a9c004e8596687ccc7' --
data-dir=/tmp/etcd2_backup/ -advertise-client-urls http://172.17.8.102:2379 -initial-advertise-peer-url
s http://172.17.8.102:2380 -listen-client-urls  http://0.0.0.0:2379 -listen-peer-urls http://172.17.8.1
02:2380
```

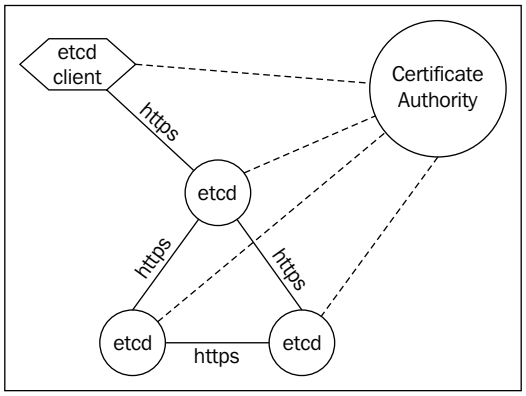
There are two approaches to handle the migration:

- Add a new member and remove the old member. We can use `etcdctl member add` and `etcdctl member remove`.
- Make a copy of the etcd database, move it to the new node, and update it.

With the first approach, the new member has a different identity. With the second approach, we can have the new node retain the same old identity. With the first approach, there is no need to stop the etcd service, while we need to stop the etcd service before taking the backup in the second approach.

Etcd security

A secure etcd is needed to ensure that the client-to-server communication and server-to-server communication are secure. The following figure shows you the different components involved in providing etcd security. Certificate authority is used to provide and verify certificates for the etcd client-to-server and server-to-server communication:



Certificate authority – etcd-ca

Certificate authority is a trusted source that issues certificates to a trusted server. Other than using standard **certificate authorities** (**CA**), etcd allows for a custom CA. Etcd-ca (`https://github.com/coreos/etcd-ca`) is a GO application that can be used as a CA for testing purposes. Recently, etcd has migrated to CFSSL (`https://github.com/cloudflare/cfssl`) as the official tool for certificates.

Installing etcd-ca

I installed etcd-ca in my Linux VM running Ubuntu 14.04 using the following steps:

```
git clone https://github.com/coreos/etcd-ca
cd etcd-ca
./build
```

Note: The GO application needs to be installed before the etcd-ca installation.

Following three steps are needed to setup etcd-ca:

1. Creating a CA using etcd-ca.
2. Creating server keys.
3. Creating client keys.

The following command, `etcd-ca init`, is used to create a CA. This is a one-time procedure. The following screenshot shows you the output when creating a CA:

```
smakam14@jungle1:~$ etcd-ca init
Enter passphrase (empty for no passphrase):
Enter same passphrase again:
Created ca/key
Created ca/crt
```

The following commands can be used to create a server certificate:

```
etcd-ca new-cert -ip  172.17.8.101 core-01
etcd-ca sign core-01
etcd-ca chain core-01
etcd-ca export --insecure core-01 | tar xvf -
```

In the preceding command, `172.17.8.101` is the CoreOS node IP and `core-01` is the node name. These steps will create `core-01.crt` and `core-01.key.insecure`.

The following commands can be used to create a client certificate:

```
etcd-ca new-cert -ip 192.168.56.104 client
etcd-ca sign client
etcd-ca chain client
etcd-ca export --insecure client | tar xvf -
```

In the preceding command, `192.168.56.104` is the client node IP. These steps will create `client.crt` and `client.key.insecure`.

Etcd secure client-to-server communication using a server certificate

A server certificate is used by the client to ensure the server's identity. The following command starts the etcd server using a server certificate and the key that was generated in the previous section:

```
etcd2 -name core-01 --cert-file=/home/core/core-01.crt --key-file=/home/
core/core-01.key  --advertise-client-urls=https://172.17.8.101:2379
--listen-client-urls=https://172.17.8.101:2379
```

The following is an example to set a key and retrieve it using a secure mechanism:

```
curl --cacert /home/smakam14/.etcd-ca/ca.crt https://172.17.8.101:2379/
v2/keys/foo -XPUT -d value=bar -v
curl --cacert /home/smakam14/.etcd-ca/ca.crt https://172.17.8.101:2379/
v2/keys/foo
```

The following example uses etcdctl to do the key retrieval:

```
etcdctl --ca-file /home/smakam14/.etcd-ca/ca.crt --peers
https://172.17.8.101:2379 get /foo
```

Etcd secure client-to-server communication using server certificate and client certificate

In the previous example, only the server had a certificate. In this example, we will generate a client certificate so that the server can verify the client's identity. The following command starts the etcd server using a server certificate and key and enabling client authentication. The server certificate and keys are the same as generated in the previous section:

```
etcd2 -name core-01 --data-dir=core-01 -client-cert-auth -trusted-ca-
file=/home/core/ca.crt -cert-file=/home/core/key.crt  -key-file=/home/
core/key.key -advertise-client-urls https://172.17.8.101:2379 -listen-
client-urls https://172.17.8.101:2379
```

The following is an example to set a key and retrieve it using a secure client and server mechanism. The client certificate and key are the same as generated in the previous section:

```
curl --cacert /home/smakam14/.etcd-ca/ca.crt --cert /home/
smakam14/client.crt --key /home/smakam14/client.key.insecure -L
https://172.17.8.101:2379/v2/keys/foo -XPUT -d value=bar -v
curl --cacert /home/smakam14/.etcd-ca/ca.crt --cert /home/
smakam14/client.crt --key /home/smakam14/client.key.insecure
https://172.17.8.101:2379/v2/keys/foo
```

The following example uses etcdctl to do the key retrieval:

```
etcdctl --ca-file /home/smakam14/.etcd-ca/ca.crt --cert-file /home/
smakam14/client.crt --key-file /home/smakam14/client.key.insecure --peers
https://172.17.8.101:2379 get /foo
```

A secure cloud-config

The following is a sample `cloud-config` that sets up the etcd security environment variables as well as the necessary certificate and keys:

```
cloud-config
write_files:

 - path: /run/systemd/system/etcd2.service.d/30-configuration.conf
   permissions: 0644
   content: |
     [Service]
     Environment=ETCD_NAME=core-01
     Environment=ETCD_VERBOSE=1
     # Encryption
     Environment=ETCD_CLIENT_CERT_AUTH=1
     Environment=ETCD_TRUSTED_CA_FILE=/home/core/ca.crt
     Environment=ETCD_CERT_FILE=/home/core/server.crt
     Environment=ETCD_KEY_FILE=/home/core/server.key

 - path: /home/core/ca.crt
   permissions: 0644
   content: |
     -----BEGIN CERTIFICATE-----
      -----END CERTIFICATE-----

 - path: /home/core/server.crt
   permissions: 0644
   content: |
     -----BEGIN CERTIFICATE-----
     -----END CERTIFICATE-----

 - path: /home/core/server.key
   permissions: 0644
   content: |
     -----BEGIN RSA PRIVATE KEY-----
     -----END RSA PRIVATE KEY-----

coreos:
 etcd2:
    # Static cluster
    initial-cluster-token: etcd-cluster-1
    initial-cluster: core-01=http://$private_ipv4:2380
    initial-cluster-state: new
```

```
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    # listen on both the official ports and the legacy ports
    # legacy ports can be omitted if your application doesn't depend
on them
    listen-client-urls: http://$public_ipv4:2379
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
 units:
   - name: etcd2.service
     command: start
```

Authentication

Before the introduction of the authentication feature, there were no restrictions on access to the etcd database. The authentication feature was introduced as an experimental feature in etcd 2.1.0 and allows access to a specific set of keys based on the username and password.

There are two entities associated with authentication:

- **Users**: Users can be created with a username and password. Before enabling the authentication feature, a root user needs to be created. The root user has substantially more privileges/permissions to add users and roles and assign role permissions.
- **Roles**: Roles can be used to restrict access to a specific key or directory that holds multiple keys. Roles are assigned to users, and manipulations of the keys can be done based on the username.

To get started with authentication, we need to first create a root user and then enable authentication.

Create a root user first, as shown in the following screenshot:

```
core@core-01 ~ $ etcdctl user add root
New password:
User root created
```

Enable authentication as follows:

```
core@core-01 ~ $ etcdctl auth enable
Authentication Enabled
```

The following example illustrates the etcd authentication.

Let's create a sample keyset, user, and role:

1. Create `/dir1/key1` and `/dir2/key2` keys.
2. Create a `role_dir1` role that has access to `/dir1/*` only.
3. Create a `role_dir2` role that has access to `/dir2/*` only.
4. Create `user1` and grant the `role_dir1` role.
5. Create `user2` and grant the `role_dir2` role.

At this point, `user1` will be able to access `/dir1/*` only and `user2` will be able to access `/dir2/*` only.

The following is a breakdown of the steps:

Create `/dir1/key1` and `/dir2/key2` keys:

```
core@core-01 ~ $ etcdctl set /dir/key1 value1
value1
```

```
core@core-01 ~ $ etcdctl set dir2/key1 value1
value1
```

Create a `role_dir1` role that has access to `/dir1/*` only:

```
core@core-01 ~ $ etcdctl -u root:test role add role_dir1
Role role_dir1 created
```

```
core@core-01 ~ $ etcdctl -u root:test role grant role_dir1 -path '/dir1/*' -readwrite
Role role_dir1 updated
```

Create a `role_dir2` role that has access to `/dir2/*` only:

```
core@core-01 ~ $ etcdctl -u root:test role add role_dir2
Role role_dir2 created
```

```
core@core-01 ~ $ etcdctl -u root:test role grant role_dir2 -path '/dir2/*' -readwrite
Role role_dir2 updated
```

Create `user1` and grant the `role_dir1` role:

```
core@core-01 ~ $ etcdctl -u root:test user add user1
New password:
User user1 created
```

```
core@core-01 ~ $ etcdctl -u root:test user grant user1 -roles role_dir1
User user1 updated
```

Create `user2` and grant the `role_dir2` role:

```
core@core-01 ~ $ etcdctl -u root:test user add user2
New password:
User user2 created
```

```
core@core-01 ~ $ etcdctl -u root:test user grant user2 -roles role_dir2
User user2 updated
```

Now, we can verify that `user1` has access only to `/dir1/key1`. As shown in the following screenshot, `user1` is not able to access `/dir2/key1`:

```
core@core-01 ~ $ etcdctl -u user1:test get dir1/key1
value1
core@core-01 ~ $ etcdctl -u user1:test get dir2/key1
Error:  110: The request requires user authentication (Insufficient credentials) [0]
```

Similarly, `user2` has access only to `/dir2/key1`:

```
core@core-01 ~ $ etcdctl -u user2:test get dir1/key1
Error:  110: The request requires user authentication (Insufficient credentials) [0]
core@core-01 ~ $ etcdctl -u user2:test get dir2/key1
value1
```

Etcd debugging

Etcd log files can be checked using the following command:

```
Journalctl -u etcd2.service
```

Default logging is set to `INFO`. For more elaborate logging, we can set `ETCD_DEBUG=1` in the environment file or use the `-debug` command-line option.

Sometimes, it's useful to check the curl command associated with the etcdctl CLI command. This can be achieved using the `--debug` option. The following is an example:

```
core@core-01 ~ $ etcdctl --debug set /messsage hello
Cluster-Endpoints: http://172.17.8.103:2379, http://172.17.8.101:2379, http://172.17.8.102:2379
Curl-Example: curl -X PUT http://172.17.8.102:2379/v2/keys/messsage -d value=hello
hello
```

Systemd

An overview of Systemd was provided in *Chapter 1, CoreOS Overview*. Systemd is the init system used by CoreOS and is always on by default. In this section, we will walk through some of the Systemd internals.

Unit types

Units describe a particular task along with its dependencies and the execution order. Some units are started on the CoreOS system by default. CoreOS users can also start their own units. System-started units are at `/usr/lib64/systemd/system` and user-started units are at `/etc/systemd/system`.

The following are some of the common unit types:

- **Service unit**: This is used to start a particular daemon or process. Examples are `sshd.service` and `docker.service`. The `sshd.service` unit starts the SSH service, and `docker.service` starts the docker daemon.
- **Socket unit**: This is used for local IPC or network communication. Examples are `systemd-journald.socket` and `docker.socket`. There is a corresponding service associated with a socket that manages the socket. For example, `docker.service` manages `docker.socket`. In `docker.service`, `docker.socket` is mentioned as a dependency. `Docker.socket` provides remote connectivity to the docker engine.
- **Target unit**: This is used mainly to group related units so that they can be started together. All user-created services are in `multi-user.target`.

- **Mount unit**: This is used to mount disks to the filesystem. Examples are `tmp.mount` and `usr-share-oem.mount`. The following is a relevant section of `usr-share-oem.mount` that mounts `/usr/share/oem`:

```
[Mount]
What=/dev/disk/by-label/OEM
Where=/usr/share/oem
Options=nodev,commit=600
Type=ext4
```

- **Timer unit**: These are units that are started periodically based on the interval specified. Examples are `update-engine-stub.timer` and `logrotate.timer`. The following is a relevant section of `update-engine-stub.timer`, where `update-engine-stub.service` is invoked every `41 minutes` to check for CoreOS updates:

```
[Timer]
OnBootSec=7minutes
OnActiveSec=41minutes
Unit=update-engine-stub.service
```

Unit specifiers

When writing systemd units, it is useful to access system environment variables such as hostname, username, IP address, and so on so that we can avoid hardcoding and use the same systemd unit across systems. For this, systemd provides you with unit specifiers, which are shortcuts to get to the system environment.

The following are some common unit specifiers:

- `%H`: Hostname
- `%m`: Machine ID
- `%u`: Username

A complete list of unit specifiers is specified at `http://www.freedesktop.org/software/systemd/man/systemd.unit.html#Specifiers`.

The following service example illustrates the usage of unit specifiers. In this example, we are setting the key-value pair associated with different specifiers in etcd in ExecStartPre. In ExecStartPost, we are getting the key-value and then cleaning up in the end:

```
[Unit]
Description=My Service

[Service]
KillMode=none
ExecStartPre=/usr/bin/etcdctl set hostname %H ; /usr/bin/etcdctl set
machinename %m ; /usr/bin/etcdctl set bootid %b ; /usr/bin/etcdctl set
unitname %n ; /usr/bin/etcdctl set username %u
ExecStart=/bin/echo hello, set done, will echo and remove
ExecStartPost=/usr/bin/etcdctl get hostname ; /usr/bin/etcdctl get
machinename ; /usr/bin/etcdctl get bootid ; /usr/bin/etcdctl get
unitname ; /usr/bin/etcdctl get username ;
ExecStartPost=/usr/bin/etcdctl rm hostname ; /usr/bin/etcdctl rm
machinename ; /usr/bin/etcdctl rm bootid ; /usr/bin/etcdctl rm
unitname ; /usr/bin/etcdctl rm username ;

[Install]
WantedBy=multi-user.target
```

To execute this service, it is necessary to execute all the following operations with sudo:

1. Create the `unitspec.service` file in `/etc/systemd/system` with the preceding content.
2. Enable the service with `systemctl enable unitspec.service`.
3. Start the service with `systemctl start unitspec.service`.
4. If we change the service after this, we need to execute command `systemctl daemon-reload` before starting the service.

The following are the `journalctl` logs associated with the service where we can see the key being set and retrieved and the corresponding unit specifier value:

`journalctl -u unitspec.service`

```
Sep 27 04:38:22 core-01 systemd[1]: Started My Service.
Sep 27 04:40:14 core-01 systemd[1]: Starting My Service...
Sep 27 04:40:14 core-01 etcdctl[5405]: core-01
Sep 27 04:40:14 core-01 etcdctl[5412]: d260aa14b9bf474a96e01b7591bbc017
Sep 27 04:40:14 core-01 etcdctl[5419]: a649809fbc43406280b63d41d9f76705
Sep 27 04:40:14 core-01 etcdctl[5430]: unitspec.service
Sep 27 04:40:14 core-01 etcdctl[5439]: root
Sep 27 04:40:14 core-01 echo[5446]: hello, set done, will echo and remove
Sep 27 04:40:14 core-01 etcdctl[5447]: core-01
Sep 27 04:40:14 core-01 etcdctl[5455]: d260aa14b9bf474a96e01b7591bbc017
Sep 27 04:40:14 core-01 etcdctl[5463]: a649809fbc43406280b63d41d9f76705
Sep 27 04:40:14 core-01 etcdctl[5471]: unitspec.service
Sep 27 04:40:14 core-01 etcdctl[5480]: root
```

Unit templates

Systemd units can be created as a template, and the same template unit can be used to instantiate multiple units based on the invocation of templates.

Templates are created as `unitname@.service`. The invocation of templates can be done using `unitname@instanceid.service`. In the unit file, the `unit name` can be accessed with `%p` and `instanceid` can be accessed using `%i`.

The following is an example template file, `unitspec@.service`:

```
[Unit]
Description=My Service

[Service]
ExecStartPre=/usr/bin/etcdctl set instance%i %i ; /usr/bin/etcdctl set
prefix %p
ExecStart=/bin/echo Demonstrate systemd template

[Install]
WantedBy=multi-user.target
```

To execute this service, it is necessary to execute all the following operations with sudo:

1. Create the `unitspec@.service` file in `/etc/systemd/system`.
2. Enable the service with `systemctl enable unitspec@.service`.

3. Start multiple services with `systemctl start unitspec@1.service` and `systemctl start unitspec@2.service`.

If we look at the etcd content, we can see that the instance value gets updated based on the `%i` argument supplied in the unit name and creates the `instance1` and `instance2` keys:

```
core@core-01 /etc/systemd/system $ etcdctl ls / --recursive
/coreos.com
/coreos.com/updateengine
/coreos.com/updateengine/rebootlock
/coreos.com/updateengine/rebootlock/semaphore
/prefix
/instance1
/instance2
core@core-01 /etc/systemd/system $ etcdctl get instance1
1
core@core-01 /etc/systemd/system $ etcdctl get instance2
2
```

The following example gives a more practical example of instantiated units. It uses a template nginx service, `nginx@.service`, where the port number of the web service is passed dynamically:

```
[Unit]
Description=Apache web server service
After=etcd.service
After=docker.service

[Service]
TimeoutStartSec=0
Restart=always
EnvironmentFile=/etc/environment
ExecStartPre=-/usr/bin/docker kill nginx%i
ExecStartPre=-/usr/bin/docker rm nginx%i
ExecStartPre=/usr/bin/docker pull nginx
ExecStart=/usr/bin/docker run --name nginx%i -p ${COREOS_PUBLIC_
IPV4}:%i:80 nginx
ExecStop=/usr/bin/docker stop nginx%i

[Install]
WantedBy=multi-user.target
```

There are two service options used in the preceding code:

- `Timeoutstartsec`: This specifies the time taken to start the service, and if the service is not started by this time, it gets killed. The `none` parameter disables this option and is useful when downloading big containers.
- `Restart`: This controls the restartability of the service. Here we have specified `always` to restart the service in case there is a failure associated with this service.

Let's create two instances of this service using the following commands:

```
Sudo systemctl enable nginx@.service
Sudo systemctl start nginx@8080.service
Sudo systemctl start nginx@8081.service
```

This creates two docker containers with nginx service; the first one exposing port `8080` and the second one exposing port `8081`.

Let's look at `docker ps`:

```
core@core-01 ~ $ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                                  NAMES
b5bc63e33780        nginx               "nginx -g 'daemon of   3 minutes ago       Up 3 minutes        443/tcp, 172.17.8.101:8081->80/tcp     nginx8081

d110739bfce7        nginx               "nginx -g 'daemon of   3 minutes ago       Up 3 minutes        443/tcp, 172.17.8.101:8080->80/tcp     nginx8080
```

Let's look at the status of the two units. As we can see in the following screenshot, the units are in an active (running) state:

```
core@core-01 ~ $ systemctl status nginx@8080.service
? nginx@8080.service - Apache web server service
   Loaded: loaded (/etc/systemd/system/nginx@.service; disabled; vendor preset: disabled)
   Active: active (running) since Sun 2015-09-27 07:00:29 UTC; 4min 58s ago
```

```
core@core-01 ~ $ systemctl status nginx@8081.service
? nginx@8081.service - Apache web server service
   Loaded: loaded (/etc/systemd/system/nginx@.service; disabled; vendor preset: disabled)
   Active: active (running) since Sun 2015-09-27 07:00:56 UTC; 4min 34s ago
```

Drop-in units

Drop-in units are useful to change system unit properties at runtime. There are four ways of creating drop-in units.

Default cloud-config drop-in units

Parameters specified in the `cloud-config` user data will automatically be configured as drop-in units. For example, let's look at the `etcd2.service cloud-config`:

```
coreos:
  etcd2:
    #generate a new token for each unique cluster from https://discovery.etcd.io/new
    discovery: https://discovery.etcd.io/c63c47bacd8edf8fe65150201b8e12f0
    # multi-region and multi-cloud deployments need to use $public_ipv4
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    # listen on both the official ports and the legacy ports
    # legacy ports can be omitted if your application doesn't depend on them
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_ipv4:7001
```

Let's look at the `etcd2.service` status:

```
core@core-02 /etc/systemd/system/fleet.service.d $ systemctl status etcd2.service
? etcd2.service - etcd2
   Loaded: loaded (/usr/lib64/systemd/system/etcd2.service; disabled; vendor preset: disabled)
  Drop-In: /run/systemd/system/etcd2.service.d
           └─20-cloudinit.conf
   Active: active (running) since Sat 2015-09-26 15:31:03 UTC; 16h ago
```

As we can see in the preceding output, the default drop-in unit is `20-cloudinit.conf`.

`20-cloudinit.conf` will contain the parameters specified in `etcd2 cloud-config` as environment variables, as shown in the following screenshot:

```
core@core-02 /etc/systemd/system/fleet.service.d $ cat /run/systemd/system/etcd2.service.d/20-cloudinit.conf
[Service]
Environment="ETCD_ADVERTISE_CLIENT_URLS=http://172.17.8.102:2379"
Environment="ETCD_DISCOVERY=https://discovery.etcd.io/c63c47bacd8edf8fe65150201b8e12f0"
Environment="ETCD_INITIAL_ADVERTISE_PEER_URLS=http://172.17.8.102:2380"
Environment="ETCD_LISTEN_CLIENT_URLS=http://0.0.0.0:2379,http://0.0.0.0:4001"
Environment="ETCD_LISTEN_PEER_URLS=http://172.17.8.102:2380,http://172.17.8.102:7001"
```

Cloud-config custom drop-in units

We can specify the drop-in unit as part of the `cloud-config`. The following is an example of the `fleet.service` drop-in unit, where we change the default `Restart` parameter from `Always` to `No`:

```
units:
  # To use etcd2, comment out the above service and uncomment these
  # Note: this requires a release that contains etcd2
  - name: etcd2.service
    command: start
  - name: fleet.service
    drop-ins:
      - name: norestart.conf
        content: |
          [Service]
          Restart=no
    command: start
```

When we use this `cloud-config`, the `norestart.conf` drop-in file gets automatically created as can be seen from the `fleet.service` status:

```
core@core-01 ~ $ systemctl status fleet.service
? fleet.service - fleet daemon
   Loaded: loaded (/usr/lib64/systemd/system/fleet.service; disabled; vendor preset: disabled)
  Drop-In: /run/systemd/system/fleet.service.d
           └─20-cloudinit.conf
         /etc/systemd/system/fleet.service.d
           └─norestart.conf
   Active: active (running) since Sun 2015-09-27 11:46:49 UTC; 15s ago
```

This configuration change will keep `fleet.service` non-restartable.

Runtime drop-in unit – specific parameters

We can change specific properties of the service using the drop-in configuration file. The following is the service section of `fleet.service`, which shows the default parameters:

```
[Service]
ExecStart=/usr/bin/fleetd
Restart=always
RestartSec=10s
```

This specifies that the service needs to be started in 10 seconds in case the service dies because of some error. Let's check whether the restart works by killing the Fleet service.

We can kill the service as follows:

```
Sudo kill -9 <fleet pid>
```

The following is a log showing the Fleet service restarting in 10 seconds, which is due to `Restartsec` specified in the service configuration:

```
Sep 27 09:17:45 core-02 systemd[1]: fleet.service: Main process exited, code=killed, status=9/KILL
Sep 27 09:17:45 core-02 systemd[1]: fleet.service: Unit entered failed state.
Sep 27 09:17:45 core-02 systemd[1]: fleet.service: Failed with result 'signal'.
Sep 27 09:17:55 core-02 systemd[1]: fleet.service: Service hold-off time over, scheduling restart.
Sep 27 09:17:55 core-02 systemd[1]: Started fleet daemon.
Sep 27 09:17:55 core-02 systemd[1]: Starting fleet daemon...
```

To prove the runtime drop-in configuration change, let's create a configuration file where we disable the restart for the Fleet service.

Create `norestart.conf` under `/etc/systemd.system/system/fleet.service.d`:

```
[Service]
Restart=no
```

Now, let's restart the systemd configuration:

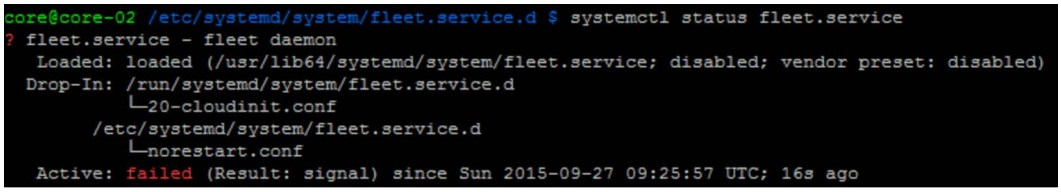
```
Sudo systemd daemon-reload
```

Let's check the status of `fleet.service` now:

```
core@core-02 /etc/systemd/system/fleet.service.d $ systemctl status fleet.service
● fleet.service - fleet daemon
   Loaded: loaded (/usr/lib64/systemd/system/fleet.service; disabled; vendor preset: disabled)
  Drop-In: /run/systemd/system/fleet.service.d
           └─20-cloudinit.conf
           /etc/systemd/system/fleet.service.d
           └─norestart.conf
   Active: active (running) since Sun 2015-09-27 09:17:55 UTC; 5min ago
```

We can see that other than `20-cloudinit.conf`, we also have a `norestart.conf` drop-in unit.

Now, if we kill the Fleet service, it does not get restarted as the restart option has been disabled by the `restart.conf` drop-in unit. `Fleet.service` stays in a failed state, as shown in the following screenshot:



Runtime drop-in unit – full service

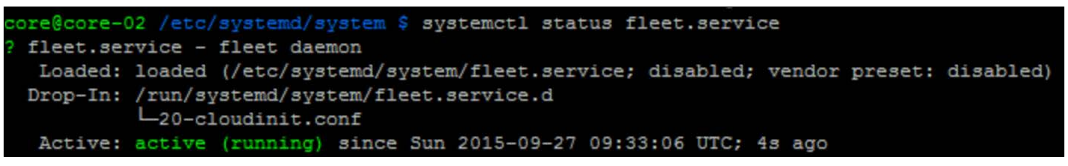
In this approach, we can replace the complete system service using our own service. Let's change the restart option by creating this `fleet.service` file in `/etc/systemd/system`:

```
[Unit]
Description=fleet daemon

After=etcd.service
After=etcd2.service

Wants=fleet.socket
After=fleet.socket

[Service]
ExecStart=/usr/bin/fleetd
Restart=no

[Install]
WantedBy=multi-user.target
```

We can start the `fleet.service` as follows:

```
Sudo systemctl start fleet.service
```

Let's see the status of `fleet.service`:



From the preceding output, we can see that `fleet.service` is picked up from `/etc/systemd/system`.

If we compare this option (a drop-in unit with a complete service change) with the previous option (a drop-in unit with a specific parameter change), the previous option gives the flexibility to change specific parameters and not touch the original set. This makes it easier to handle upgrades when new versions of the service allow additional options.

Network units

The `systemd-networkd` service manages networks. System-configured networks are specified in `/usr/lib64/systemd/network` and user-configured networks are specified in `/etcd/systemd/network`. The following is a sample Vagrant configured systemd-network file to configure the `eth1` IP address:

```
core@core-01 /etc/systemd/network $ cat 50-vagrant1.network
[Match]
Name=eth1

[Network]
Address=172.17.8.101/24
```

The `ifconfig` output associated with `eth1` shows the IP address that Vagrant configured, as shown in the following screenshot:

```
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.8.101  netmask 255.255.255.0  broadcast 172.17.8.255
        inet6 fe80::a00:27ff:fe00:e52c  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:00:e5:2c  txqueuelen 1000  (Ethernet)
```

As an example, let's try to change the `eth1` IP address. There are three steps:

1. Stop `systemd-networkd.service`.
2. Flush the IP address.
3. Set a new IP address.

Let's create a service file to flush the `eth1` IP address and another network file specifying the new IP address for `eth1`.

Create a service file to flush the `eth1` IP address as follows. We need to place this in `/etc/systemd/system/down-eth1.service`.

```
[Unit]
Description=eth1 flush

[Service]
Type=oneshot
ExecStart=/usr/bin/ip link set eth1 down
ExecStart=/usr/bin/ip addr flush dev eth1

[Install]
WantedBy=multi-user.target
```

The following is the network file to specify the `eth1` new address. We need to place this in `/etc/systemd/network/40-eth1.network`:

```
[Match]
Name=eth1

[Network]
Address=172.17.8.110/24
Gateway=172.17.8.1
```

The steps to change the IP address are as follows:

1. Stop the system-networkd service by `sudo systemctl stop systemd-networkd.service`.
2. Flush the `eth1` IP address by `sudo systemctl start down-eth1.service`.
3. Start `systemd-networkd.service` by `sudo systemctl start systemd-networkd.service`.

If we look at the ifconfig output now, we should see the new IP address `172.17.8.110`:

```
eth1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1500
        inet 172.17.8.110  netmask 255.255.255.0  broadcast 172.17.8.255
        inet6 fe80::a00:27ff:fe00:e52c  prefixlen 64  scopeid 0x20<link>
        ether 08:00:27:00:e5:2c  txqueuelen 1000  (Ethernet)
```

Fleet

Fleet is a cluster manager/scheduler that controls service creation at the CoreOS cluster level. We can think of Fleet as Systemd for the cluster. For an overview of Fleet, refer to *Chapter 1*, *CoreOS Overview*. Fleet is used mainly for the orchestration of critical system services, while other orchestration solutions such as Kubernetes are used for application service orchestration. Fleet is not under active development and is mostly under the maintenance mode.

Installation

Fleet is installed and started by default in CoreOS. The following is the Fleet version in the CoreOS stable 766.3.0 release:

```
core@core-01 /usr/lib64/systemd/system $ fleet --version
fleetd version 0.10.2
core@core-01 /usr/lib64/systemd/system $ fleetctl --version
fleetctl version 0.10.2
```

Fleet can also be installed in a standalone Linux machine. Fleet releases can be found at `https://github.com/coreos/fleet/releases`.

Accessing Fleet

The following are different approaches to access Fleet.

Local fleetctl

The `fleetctl` command is present in each CoreOS node and can be used to control Fleet services.

Remote fleetctl

The `fleetctl` command can be used to access non-local CoreOS nodes by specifying an endpoint argument. The following is an example:

```
Fleetctl --endpoint=http://172.17.8.101:2379 list-machines
```

```
smakam14@jungle1:~$ fleetctl --endpoint=http://172.17.8.101:2379  list-machines
MACHINE         IP              METADATA
12d30fe3...     172.17.8.103    -
91ab09f8...     172.17.8.101    -
949a5786...     172.17.8.102    -
```

Remote fleetctl with an SSH tunnel

The previous example did not use any authentication. To make access to fleetctl secure, we can use the SSH authentication scheme. It is necessary to add the CoreOS node private key to the local SSH authentication agent for this mode. For a Vagrant CoreOS cluster, the private key is stored in `~/.vagrant.d/insecure_private_key`. For an AWS CoreOS cluster, the private key can be downloaded as part of the initial key creation.

To add a private key to the authentication agent:

```
eval `ssh-agent -s`
ssh-add <private key>
```

```
smakam14@jungle1:~$ eval `ssh-agent -s`
Agent pid 3292
```

```
smakam14@jungle1:~$ ssh-add ~/.ssh/vagrant_coreos_private_key
Identity added: /home/smakam14/.ssh/vagrant_coreos_private_key (/home/smakam14/.ssh/vagrant_coreos_p
rivate key)
```

Now, we can use fleetctl to use a secure SSH to access the CoreOS cluster:

```
Fleetctl --tunnel=http://172.17.8.101 list-unit-files
```

```
smakam14@jungle1:~$ fleetctl --tunnel=172.17.8.101  list-unit-files
UNIT            HASH    DSTATE          STATE           TARGET
hello.service   0d1c468 launched        launched        12d30fe3.../172.17.8.103
```

Remote HTTP

Remote HTTP Fleet API access is disabled by default.

To enable remote access, create a `.socket` file to expose the Fleet API port. The following is an example Fleet configuration file to expose port `49153` for external API access:

```
core@core-01 /etc/systemd/system/fleet.socket.d $ cat 30-fleetexternal.conf
[Socket]
ListenStream=172.17.8.101:49153
```

It is necessary to restart the systemd, `fleet.socket`, and `fleet.service` after creating the remote API configuration file for it to take effect:

```
Sudo systemctl daemon-reload
Sudo systemctl restart fleet.socket
Sudo systemctl restart fleet.service
```

Now, we can access the remote API. The following is an example using `fleetctl` and `curl`:

```
Fleetctl --endpoint=http://172.17.8.101:49153 list-units
```

```
smakam14@jungle1:~$ fleetctl --endpoint=http://172.17.8.101:49153 list-units
UNIT            MACHINE                         ACTIVE  SUB
hello.service   949a5786.../172.17.8.102        active  running
```

The following output shows you the unit list using the Fleet HTTP API. The following curl output is truncated to show partial output:

```
Curl -s http://172.17.8.101:49153/fleet/v1/units | jq .
```

```
smakam14@jungle1:~$ curl -s http://172.17.8.101:49153/fleet/v1/units | jq .
{
  "units": [
    {
      "options": [
        {
          "value": "My Service",
          "section": "Unit",
          "name": "Description"
        },
        {
          "value": "docker.service",
          "section": "Unit",
          "name": "After"
        },
```

Using etcd security

We can also use the secure etcd approach to access Fleet. Setting up a secure etcd is covered in the section on *Etcd security*. The following example shows the `fleetctl` command with a server certificate:

```
fleetctl --debug --ca-file ca.crt  --endpoint=https://172.17.8.101:2379
list-machines
```

Templates, scheduling, and HA

Fleet supports unit specifiers and templates similar to systemd. A unit specifier provides you with shortcuts within a service file, and templates provide reusable service files. The earlier section on systemd covered details on unit specifiers and templates. *Chapter 1*, *CoreOS Overview* covered the basics of Fleet scheduling and HA.

Fleet metadata for a node can be specified in the *Fleet* section of `cloud-config`. The following example sets the Fleet node metadata for `role` as `web`. Metadata can be used in Fleet service files to control scheduling:

```
metadata: "role=services"
```

Fleet uses a pretty simple scheduling algorithm, and X-fleet options are used to specify constraints while scheduling the service. The following are the available X-fleet options:

- `MachineMetaData`: Service gets scheduled based on matching metadata.
- `MachineId`: Service gets scheduled based on the specified `MachineId`.
- `MachineOf`: Service gets scheduled based on other services running in the same node. This can be used to schedule tightly coupled services in the same node.
- `Conflict`: This option can be used to avoid scheduling conflicting services in the same node.
- `Global`: The same service gets scheduled in all the nodes of the cluster.

The following example uses unit specifiers and templates and illustrates Fleet scheduling and HA. The following are some details of the application:

- An application consists of a WordPress container and MySQL container
- The WordPress container uses the database from the MySQL container and is linked using Docker container linking
- Linking across containers is done using the `--link` option, and it works only if both containers are on the same host
- Fleet's template feature will be used to launch multiple services using a single WordPress and MySQL template, and Fleet's X-fleet constraint feature will be used to launch the related containers on the same host
- When one of the nodes in the cluster dies, Fleet's HA mechanism will take care of rescheduling the failed units, and we will see it working in this example

The MySQL template service is as follows:

```
[Unit]
Description=app-mysql

[Service]
Restart=always
RestartSec=5
ExecStartPre=-/usr/bin/docker kill mysql%i
ExecStartPre=-/usr/bin/docker rm mysql%i
ExecStartPre=/usr/bin/docker pull mysql
ExecStart=/usr/bin/docker run --name mysql%i -e MYSQL_ROOT_
PASSWORD=mysql mysql
ExecStop=/usr/bin/docker stop mysql%i
```

The WordPress template service is as follows:

```
[Unit]
Description=wordpress

[Service]
Restart=always
RestartSec=15
ExecStartPre=-/usr/bin/docker kill wordpress%i
ExecStartPre=-/usr/bin/docker rm wordpress%i
ExecStartPre=/usr/bin/docker pull wordpress
ExecStart=/usr/bin/docker run --name wordpress%i --link mysql%i:mysql
wordpress
ExecStop=/usr/bin/docker stop wordpress%i

[X-Fleet]
MachineOf=mysql@%i.service
```

The following are some notes on the service:

- We have used `%i` as an instance specifier
- WordPress has an X-fleet constraint to schedule the corresponding MySQL container in the same node

The first step is to submit the services:

```
core@core-01 ~ $ fleetctl submit wordpress\@.service mysql\@.service
```

The next step is to load each instance of the service:

```
core@core-01 ~ $ fleetctl start wordpress@1.service mysql@1.service
Unit mysql@1.service launched on 12d30fe3.../172.17.8.103
Unit wordpress@1.service launched on 12d30fe3.../172.17.8.103
core@core-01 ~ $ fleetctl start wordpress@2.service mysql@2.service
Unit mysql@2.service launched on 91ab09f8.../172.17.8.101
Unit wordpress@2.service launched on 91ab09f8.../172.17.8.101
core@core-01 ~ $ fleetctl start wordpress@3.service mysql@3.service
Unit mysql@3.service launched on 949a5786.../172.17.8.102
Unit wordpress@3.service launched on 949a5786.../172.17.8.102
```

Let's check whether all the services are running. As can be seen in the following screenshot, we have three instances of the WordPress application and the associated MySQL database:

```
core@core-01 ~ $ fleetctl list-units
UNIT                    MACHINE                         ACTIVE  SUB
mysql@1.service         12d30fe3.../172.17.8.103        active  running
mysql@2.service         91ab09f8.../172.17.8.101        active  running
mysql@3.service         949a5786.../172.17.8.102        active  running
wordpress@1.service     12d30fe3.../172.17.8.103        active  running
wordpress@2.service     91ab09f8.../172.17.8.101        active  running
wordpress@3.service     949a5786.../172.17.8.102        active  running
```

To demonstrate HA, let's kill CoreOS `node2`. This can be done by shutting down the node.

As we can see, there are only two nodes in the cluster now:

```
core@core-01 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
12d30fe3...     172.17.8.103    -
91ab09f8...     172.17.8.101    -
```

From the following new service output, we can see that the services running on the old `node2` have been moved to `node3` now as `node2` is not available:

```
core@core-01 ~ $ fleetctl list-units
UNIT                    MACHINE                         ACTIVE  SUB
mysql@1.service         12d30fe3.../172.17.8.103        active  running
mysql@2.service         91ab09f8.../172.17.8.101        active  running
mysql@3.service         12d30fe3.../172.17.8.103        active  running
wordpress@1.service     12d30fe3.../172.17.8.103        active  running
wordpress@2.service     91ab09f8.../172.17.8.101        active  running
wordpress@3.service     12d30fe3.../172.17.8.103        active  running
```

Debugging

The status of the Fleet service can be checked using `fleetctl status`. The following is an example:

```
core@core-01 ~ $ fleetctl status hello.service
? hello.service - My Service
   Loaded: loaded (/run/fleet/units/hello.service; linked-runtime; vendor preset: disabled)
   Active: active (running) since Tue 2015-09-29 10:22:22 UTC; 1h 4min ago
```

Logs of the Fleet service can be checked using `fleetctl journal`. The following is an example:

```
core@core-01 ~ $ fleetctl journal hello.service
-- Logs begin at Sun 2015-09-27 11:47:16 UTC, end at Tue 2015-09-29 11:28:34 UTC. --
Sep 29 11:28:24 core-02 docker[896]: Hello World
Sep 29 11:28:25 core-02 docker[896]: Hello World
```

For debugging and to get the REST API corresponding to the `fleetctl` command, we can use the `--debug` option as follows:

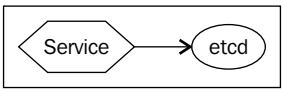
```
core@core-01 ~ $ fleetctl --debug list-unit-files
2015/09/29 11:32:42 DEBUG http.go:28: HTTP GET http://domain-sock/fleet/v1/units?alt=json
2015/09/29 11:32:42 DEBUG http.go:31: HTTP GET http://domain-sock/fleet/v1/units?alt=json 200 OK
2015/09/29 11:32:42 DEBUG http.go:28: HTTP GET http://domain-sock/fleet/v1/machines?alt=json
2015/09/29 11:32:42 DEBUG http.go:31: HTTP GET http://domain-sock/fleet/v1/machines?alt=json 200 OK
UNIT            HASH    DSTATE          STATE           TARGET
hello.service   0d1c468 launched        launched        949a5786.../172.17.8.102
```

Service discovery

Microservices are dynamic, and it is important for services to discover other services dynamically to find the IP address, port number, and metadata about the services. There are multiple schemes available to discover services, and in this section, we will cover a few schemes using etcd and Fleet for service discovery. In the later chapters of the book, we will cover advanced service discovery options.

Simple etcd-based discovery

The following figure shows you the simplest possible service discovery mechanism, where a service updates etcd with service-related details that other services can access from etcd:



The following is an example Apache service, `apacheupdateetcd@.service`, that updates the hostname and port number in etcd when the service is started:

```
[Unit]
Description=My Advanced Service
After=etcd2.service
After=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill apache%i
ExecStartPre=-/usr/bin/docker rm apache%i
ExecStartPre=/usr/bin/docker pull coreos/apache
ExecStart=/usr/bin/docker run --name apache%i -p %i:80 coreos/apache /
usr/sbin/apache2ctl -D FOREGROUND
ExecStartPost=/usr/bin/etcdctl set /domains/example.com/%H:%i running
ExecStop=/usr/bin/docker stop apache%i
ExecStopPost=/usr/bin/etcdctl rm /domains/example.com/%H:%i

[X-Fleet]
# Don't schedule on the same machine as other Apache instances
X-Conflicts=apache*@*.service
```

Let's start the service and create two instances:

```
core@core-01 ~ $ fleetctl submit apacheupdateetcd\@.service
```

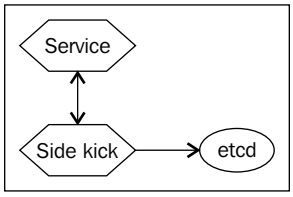
```
core@core-01 ~ $ fleetctl start apacheupdateetcd\@1.service
Unit apacheupdateetcd@1.service launched on 12d30fe3.../172.17.8.103
core@core-01 ~ $ fleetctl start apacheupdateetcd\@2.service
Unit apacheupdateetcd@2.service launched on 91ab09f8.../172.17.8.101
```

Now, we can verify that etcd gets updated with the service details of the two services:

```
/domains/example.com
/domains/example.com/core-03:1
/domains/example.com/core-01:2
```

Sidekick discovery

In the preceding scheme, there is no way to know if the service is alive and running after it has been started. The following figure shows you a slightly advanced service discovery scheme where a sidekick service updates etcd with the details of the service:



The purpose of the Side kick container is to monitor the main service and update etcd only if the Service is active. It is important to run the Side kick container in the same node as the main service that Side kick is monitoring.

The following is a Sidekick example using the Apache service and a sidekick for the Apache service.

Following is the `Apache.service` unit file:

```
[Unit]
Description=Apache web server service on port %i

# Requirements
Requires=etcd2.service
Requires=docker.service
Requires=apachet-discovery@%i.service

# Dependency ordering
After=etcd2.service
After=docker.service
Before=apachet-discovery@%i.service

[Service]
# Let processes take awhile to start up (for first run Docker
containers)
TimeoutStartSec=0

# Change killmode from "control-group" to "none" to let Docker remove
# work correctly.
KillMode=none
```

```
# Get CoreOS environmental variables
EnvironmentFile=/etc/environment

# Pre-start and Start
## Directives with "=-" are allowed to fail without consequence
ExecStartPre=-/usr/bin/docker kill apachet.%i
ExecStartPre=-/usr/bin/docker rm apachet.%i
ExecStartPre=/usr/bin/docker pull coreos/apache
ExecStart=/usr/bin/docker run --name apachet.%i -p ${COREOS_PUBLIC_
IPV4}:%i:80 coreos/apache /usr/sbin/apache2ctl -D FOREGROUND

# Stop
ExecStop=/usr/bin/docker stop apachet.%i
```

Following is the Apache sidekick service unit file:

```
[Unit]
Description=Apache Sidekick

# Requirements
Requires=etcd2.service
Requires=apachet@%i.service

# Dependency ordering and binding
After=etcd2.service
After=apachet@%i.service
BindsTo=apachet@%i.service

[Service]

# Get CoreOS environmental variables
EnvironmentFile=/etc/environment

# Start
## Test whether service is accessible and then register useful
information
ExecStart=/bin/bash -c '\
  while true; do \
    curl -f ${COREOS_PUBLIC_IPV4}:%i; \
    if [ $? -eq 0 ]; then \
      etcdctl set /services/apachet/${COREOS_PUBLIC_IPV4} \'{"host":
"%H", "ipv4_addr": ${COREOS_PUBLIC_IPV4}, "port": %i}\' --ttl 30; \
    else \
      etcdctl rm /services/apachet/${COREOS_PUBLIC_IPV4}; \
```

```
      fi; \
      sleep 20; \
    done'

# Stop
ExecStop=/usr/bin/etcdctl rm /services/apachet/${COREOS_PUBLIC_IPV4}

[X-Fleet]
# Schedule on the same machine as the associated Apache service
X-ConditionMachineOf=apachet@%i.service
```

The preceding **Side kick** container service does a periodic ping to the main service and updates the **etcd** output. If the main service is not reachable, the service-related details are removed from **etcd**.

Let's start two instances of the service:

```
core@core-01 ~ $ fleetctl list-units
UNIT                            MACHINE                         ACTIVE  SUB
apachet-discovery@1.service     12d30fe3.../172.17.8.103        active  running
apachet-discovery@2.service     91ab09f8.../172.17.8.101        active  running
apachet@1.service               12d30fe3.../172.17.8.103        active  running
apachet@2.service               91ab09f8.../172.17.8.101        active  running
```

Let's see the etcd output. As shown in the following screenshot, etcd reflects the two nodes where Apache is running:

```
/services
/services/apachet
/services/apachet/172.17.8.103
/services/apachet/172.17.8.101
```

Let's see the docker output in node1:

```
core@core-01 ~ $ docker ps
CONTAINER ID        IMAGE               COMMAND                CREATED             STATUS              P
fa0b7bdcbf12        coreos/apache       "/usr/sbin/apache2ct   38 minutes ago      Up 38 minutes       1
72.17.8.101:2->80/tcp   apachet.2
```

To demonstrate the sidekick service, let's stop the docker container and check whether the sidekick service updates etcd in order to remove the appropriate service:

```
core@core-01 ~ $ docker stop apachet.2
apachet.2
```

Let's check the status of the units. As can be seen below, `apachet@2.service` has failed and the associated sidekick service `apachet-discovery@2.service` is inactive.

```
core@core-01 ~ $ fleetctl list-units
UNIT                            MACHINE                         ACTIVE          SUB
apachet-discovery@1.service     12d30fe3.../172.17.8.103        active          running
apachet-discovery@2.service     91ab09f8.../172.17.8.101        inactive        dead
apachet@1.service               12d30fe3.../172.17.8.103        active          running
apachet@2.service               91ab09f8.../172.17.8.101        failed          failed
```

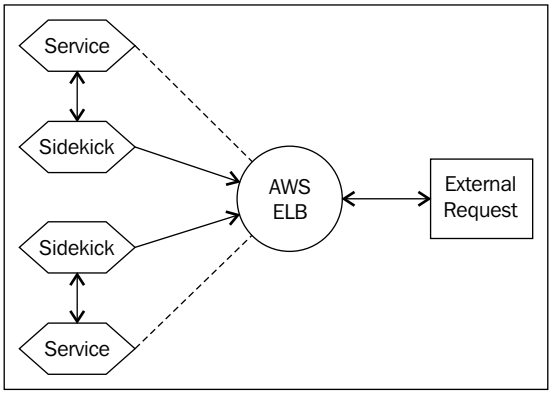
From the following output, we can see that the `apachet@2.service` details are removed from etcd:

```
/services
/services/apachet
/services/apachet/172.17.8.103
```

ELB service discovery

This is a variation of the Sidekick discovery in which, instead of Sidekick updating etcd, Sidekick updates the IP address to the load balancer and the load balancer redirects the web query to the active nodes. In this example, we will use the AWS Elastic load balancer and CoreOS elb-presence container available in the Quay repository. The elb-presence container takes care of checking the health of the nginx container and updates AWS ELB with the container's IP address.

The following figure shows you a high-level architecture of this approach:



The first step is to create ELB in AWS, as shown in the following screenshot. Here we have used AWS CLI to create ELB, `testlb`:

```
smakam14@jungle1:~$ aws elb create-load-balancer --load-balancer-name testlb --listeners "Protocol=H
TTP,LoadBalancerPort=80,InstanceProtocol=HTTP,InstancePort=80" --availability-zones us-west-2a
{
    "DNSName": "testlb-1322271991.us-west-2.elb.amazonaws.com"
}
```

We need to use `testlb` created in the preceding screenshot in the Sidekick service.

Following is the `nginx.service` unit file:

```
[Unit]
Description=nginx

[Service]
ExecStartPre=-/usr/bin/docker kill nginx-%i
ExecStartPre=-/usr/bin/docker rm nginx-%i
ExecStart=/usr/bin/docker run --rm --name nginx-%i -p 80:80 nginx
ExecStop=/usr/bin/docker stop nginx-%i

[X-Fleet]
Conflicts=nginx@*.service
```

Following is the nginx sidekick service that updates AWS ELB based on the health of `nginx.service`:

```
[Unit]
Description=nginx presence service
BindsTo=nginx@%i.service

[Service]
ExecStartPre=-/usr/bin/docker kill nginx-presence-%i
ExecStartPre=-/usr/bin/docker rm nginx-presence-%i
ExecStart=/usr/bin/docker run --rm --name nginx-presence-%i -e AWS_
ACCESS_KEY=<key> -e AWS_SECRET_KEY=<secretkey> -e AWS_REGION=us-west-2
-e ELB_NAME=testlb quay.io/coreos/elb-presence
ExecStop=/usr/bin/docker stop nginx-presence-%i

[X-Fleet]
MachineOf=nginx@%i.service
```

The following is the Fleet status after creating two instances of the service:

```
core@ip-172-31-26-79 ~ $ fleetctl list-units
UNIT                            MACHINE                         ACTIVE  SUB
nginx-presence@1.service        10c0ba1c.../172.31.26.78        active  running
nginx-presence@2.service        5df46109.../172.31.26.79        active  running
nginx@1.service                 10c0ba1c.../172.31.26.78        active  running
nginx@2.service                 5df46109.../172.31.26.79        active  running
```

As we can see in the following screenshot, AWS ELB has both the instances registered, and it will load-balance between these two instances:

```
smakam14@jungle1:~$ aws elb describe-load-balancers | grep -A 10 Instances
            "Instances": [
                {
                    "InstanceId": "i-63d287a5"
                },
                {
                    "InstanceId": "i-64d287a2"
                }
            ],
```

At this point, if we stop any instance of the nginx service, the Sidekick service will take care of removing this instance from ELB.

Summary

In this chapter, we covered the internals of Etcd, Systemd, and Fleet with sufficient hands-on examples, which will allow you to get comfortable with configuring and using these services. By keeping the development of the critical services open source, CoreOS has encouraged the usage of these services outside CoreOS as well. We also covered the basic service discovery options using Etcd, Systemd, and Fleet. In the next chapter, we will cover container networking and Flannel.

References

- Etcd docs: `https://coreos.com/etcd/docs/latest/`
- Fleet docs: `https://coreos.com/fleet/docs/latest/`
- Systemd docs: `http://www.freedesktop.org/wiki/Software/systemd/`
- Fleet service discovery: `https://coreos.com/fleet/docs/latest/examples/service-discovery.html`
- Etcd-ca: `https://github.com/coreos/etcd-ca`
- Etcd security: `https://github.com/coreos/etcd/blob/master/Documentation/security.md`

Further reading and tutorials

- Etcd security and authentication: `http://thepracticalsysadmin.com/etcd-2-1-1-encryption-and-authentication/`
- Etcd administration: `https://github.com/coreos/etcd/blob/master/Documentation/admin_guide.md`
- Why systemd: `http://blog.jorgenschaefer.de/2014/07/why-systemd.html`
- Comparing init systems: `http://centos-vn.blogspot.in/2014/06/daemon-showdown-upstart-vs-runit-vs.html`
- Systemd talk by the Systemd creator: `https://www.youtube.com/watch?v=VIPonFvPlAs`
- Service discovery overview: `http://www.gomicro.services/articles/service-discovery-overview` and `http://progrium.com/blog/2014/07/29/understanding-modern-service-discovery-with-docker/`
- Highly available Docker services using CoreOS and Consul: `http://blog.xebia.com/2015/03/24/a-high-available-docker-container-platform-using-coreos-and-consul/` and `http://blog.xebia.com/2015/04/23/how-to-deploy-high-available-persistent-docker-services-using-coreos-and-consul/`

5
CoreOS Networking and Flannel Internals

Microservices increased the need to have lots of containers and also connectivity between containers across hosts. It is necessary to have a robust Container networking scheme to achieve this goal. This chapter will cover the basics of Container networking with a focus on how CoreOS does Container networking with Flannel. Docker networking and other related container networking technologies will also be covered. The following topics will be covered in this chapter:

- Container networking basics
- Flannel internals
- A CoreOS Flannel cluster using Vagrant, AWS, and GCE
- Docker networking and experimental Docker networking
- Docker networking using Weave and Calico
- Kubernetes networking

Container networking basics

The following are the reasons why we need Container networking:

- Containers need to talk to the external world.
- Containers should be reachable from the external world so that the external world can use the services that Containers provide.
- Containers need to talk to the host machine. An example can be sharing volumes.

- There should be inter-container connectivity in the same host and across hosts. An example is a WordPress container in one host talking to a MySQL container in another host.

Multiple solutions are currently available to interconnect Containers. These solutions are pretty new and actively under development. Docker, until release 1.8, did not have a native solution to interconnect Containers across hosts. Docker release 1.9 introduced a Libnetwork-based solution to interconnect containers across hosts as well as perform service discovery. CoreOS is using Flannel for container networking in CoreOS clusters. There are projects such as Weave and Calico that are developing Container networking solutions, and they plan to be a networking container plugin for any Container runtime such as Docker or Rkt.

Flannel

Flannel is an open source project that provides a Container networking solution for CoreOS clusters. Flannel can also be used for non-CoreOS clusters. Kubernetes uses Flannel to set up networking between the Kubernetes pods. Flannel allocates a separate subnet for every host where a Container runs, and the Containers in this host get allocated an individual IP address from the host subnet. An overlay network is set up between each host that allows Containers on different hosts to talk to each other. In *Chapter 1*, *CoreOS Overview* we provided an overview of the Flannel control and data path. This section will delve into the Flannel internals.

Manual installation

Flannel can be installed manually or using the `systemd` unit, `flanneld.service`. The following command will install Flannel in the CoreOS node using a container to build the flannel binary. The flanneld Flannel binary will be available in `/home/core/flannel/bin` after executing the following commands:

```
git clone https://github.com/coreos/flannel.git
docker run -v /home/core/flannel:/opt/flannel -i -t google/golang /bin/bash -c "cd /opt/flannel && ./build"
```

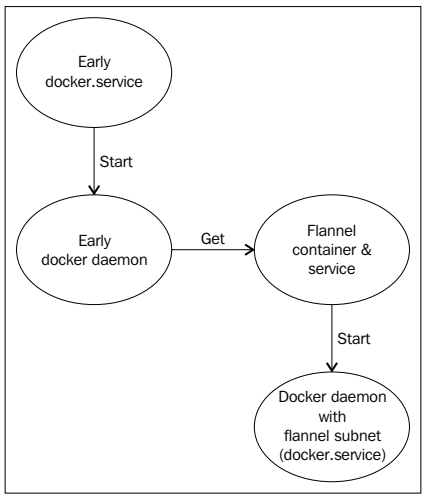
The following is the Flannel version after we build flannel in our CoreOS node:

```
core@core-01 ~ $ flanneld --version
0.5.3+git
```

Installation using flanneld.service

Flannel is not installed by default in CoreOS. This is done to keep the CoreOS image size to a minimum. Docker requires flannel to configure the network and flannel requires Docker to download the flannel container. To avoid this chicken-and-egg problem, `early-docker.service` is started by default in CoreOS, whose primary purpose is to download the flannel container and start it. A regular `docker.service` starts the Docker daemon with the flannel network.

The following figure shows you the sequence in `flanneld.service`, where the early Docker daemon starts the flannel container, which, in turn starts `docker.service` with the subnet created by flannel:



The following is the relevant section of `flanneld.service` that downloads the flannel container from the Quay repository:

```
ExecStart=/usr/libexec/sdnotify-proxy /run/flannel/sd.sock \
  /usr/bin/docker run --net=host --privileged=true --rm \
  --volume=/run/flannel:/run/flannel \
  --env=NOTIFY_SOCKET=/run/flannel/sd.sock \
  --env=AWS_ACCESS_KEY_ID=${AWS_ACCESS_KEY_ID} \
  --env=AWS_SECRET_ACCESS_KEY=${AWS_SECRET_ACCESS_KEY} \
  --env-file=${FLANNEL_ENV_FILE} \
  --volume=/usr/share/ca-certificates:/etc/ssl/certs:ro \
  --volume=${ETCD_SSL_DIR}:/etc/ssl/etcd:ro \
  quay.io/coreos/flannel:${FLANNEL_VER} /opt/bin/flanneld --ip-masq=true
```

The following output shows the early docker's running containers. Early-docker will manage Flannel only:

```
core@core-01 ~ $ docker --host=unix:///var/run/early-docker.sock ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS
          PORTS               NAMES
f76bc7c29c5c        quay.io/coreos/flannel:0.5.3   "/opt/bin/flanneld --"   2 minutes ago       Up 2 min
utes                            happy_varahamihira
```

The following is the relevant section of `flanneld.service` that updates the docker options to use the subnet created by flannel:

```
# Update docker options
ExecStartPost=/usr/bin/docker run --net=host --rm -v /run:/run \
  quay.io/coreos/flannel:${FLANNEL_VER} \
  /opt/bin/mk-docker-opts.sh -d /run/flannel_docker_opts.env -i
```

The following is the content of `flannel_docker_opts.env`—in my case—after flannel was started. The address, `10.1.60.1/24`, is chosen by this CoreOS node for its containers:

```
core@core-01 /usr/lib64/systemd/system $ cat /run/flannel_docker_opts.env
DOCKER_OPT_BIP="--bip=10.1.60.1/24"
DOCKER_OPT_IPMASQ="--ip-masq=false"
DOCKER_OPT_MTU="--mtu=1472"
```

Docker will be started as part of `docker.service`, as shown in the following screenshot, with the preceding environment file:

```
ExecStart=/usr/lib/coreos/dockerd daemon --host=fd:// $DOCKER_OPTS $DOCKER_OPT_BIP $DOCKER_OPT_MTU $DOCK
ER_OPT_IPMASQ
```

Control path

There is no central controller in flannel, and it uses etcd for internode communication. Each node in the CoreOS cluster runs a flannel agent and they communicate with each other using etcd.

As part of starting the Flannel service, we specify the Flannel subnet that can be used by the individual nodes in the network. This subnet is registered with etcd so that every CoreOS node in the cluster can see it. Each node in the network picks a particular subnet range and registers atomically with etcd.

The following is the relevant section of `cloud-config` that starts `flanneld.service` along with specifying the configuration for Flannel. Here, we have specified the subnet to be used for flannel as `10.1.0.0/16` along with the encapsulation type as `vxlan`:

```
    - name: flanneld.service
      drop-ins:
        - name: 50-network-config.conf
          content: |
            [Service]
            ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config '{ "Network": "10.1.0.0/16", "B
ackend": {"Type": "vxlan"}}'
      command: start
```

The preceding configuration will create the following etcd key as seen in the node. This shows that `10.1.0.0/16` is allocated for flannel to be used across the CoreOS cluster and that the encapsulation type is `vxlan`:

```
core@core-01 ~ $ etcdctl get /coreos.com/network/config
{ "Network": "10.1.0.0/16", "Backend": {"Type": "vxlan"}}
```

Once each node gets a subnet, containers started in this node will get an IP address from the IP address pool allocated to the node. The following is the etcd subnet allocation per node. As we can see, all the subnets are in the `10.1.0.0/16` range that was configured earlier with etcd and with a 24-bit mask. The subnet length per host can also be controlled as a flannel configuration option:

```
/coreos.com/network/subnets/10.1.25.0-24
/coreos.com/network/subnets/10.1.1.0-24
/coreos.com/network/subnets/10.1.19.0-24
```

Let's look at `ifconfig` of the Flannel interface created in this node. The IP address is in the address range of `10.1.0.0/16`:

```
flannel.1: flags=4163<UP,BROADCAST,RUNNING,MULTICAST>  mtu 1450
        inet 10.1.19.0  netmask 255.255.0.0  broadcast 0.0.0.0
```

Data path

Flannel uses the Linux bridge to encapsulate the packets using an overlay protocol specified in the Flannel configuration. This allows for connectivity between containers in the same host as well as across hosts.

The following are the major backends currently supported by Flannel and specified in the JSON configuration file. The JSON configuration file can be specified in the *Flannel* section of `cloud-config`:

- **UDP**: In UDP encapsulation, packets from containers are encapsulated in UDP with the default port number `8285`. We can change the port number if needed.
- **VXLAN**: From an encapsulation overhead perspective, VXLAN is efficient when compared to UDP. By default, port `8472` is used for VXLAN encapsulation. If we want to use an IANA-allocated VXLAN port, we need to specify the port field as `4789`.
- **AWS-VPC**: This is applicable to using Flannel in the AWS VPC cloud. Instead of encapsulating the packets using an overlay, this approach uses a VPC route table to communicate across containers. AWS limits each VPC route table entry to 50, so this can become a problem with bigger clusters.

The following is an example of specifying the AWS type in the flannel configuration:

```
          ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config '{ "Network": "10.1.0.0/16" , "
Backend": {"Type": "aws-vpc"}}'
```

- **GCE**: This is applicable to using Flannel in the GCE cloud. Instead of encapsulating the packets using an overlay, this approach uses the GCE route table to communicate across containers. GCE limits each VPC route table entry to `100`, so this can become a problem with bigger clusters.

The following is an example of specifying the GCE type in the Flannel configuration:

```
          ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/config '{ "Network": "10.1.0.0/16" , "
Backend": {"Type": "gce"}}'
```

Let's create containers in two different hosts with a VXLAN encapsulation and check whether the connectivity is fine. The following example uses a Vagrant CoreOS cluster with the Flannel service enabled.

Configuration in Host1:

Let's start a `busybox` container:

```
core@core-01 ~ $ docker run -ti busybox sh
```

Let's check the IP address allotted to the container. This IP address comes from the IP pool allocated to this CoreOS node by the flannel agent. `10.1.19.0/24` was allocated to `host1` and this container got the `10.1.19.2` address:

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0A:01:13:02
          inet addr:10.1.19.2  Bcast:0.0.0.0  Mask:255.255.255.0
```

Configuration in Host2:

Let's start a `busybox` container:

```
core@core-02 ~ $ docker run -ti busybox sh
```

Let's check the IP address allotted to this container. This IP address comes from the IP pool allocated to this CoreOS node by the flannel agent. `10.1.1.0/24` was allocated to `host2` and this container got the `10.1.1.2` address:

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0A:01:01:02
          inet addr:10.1.1.2  Bcast:0.0.0.0  Mask:255.255.255.0
```

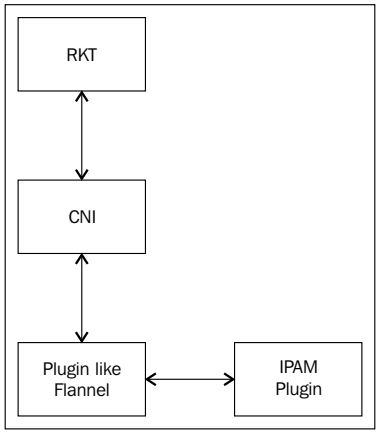
The following output shows you the ping being successful between container 1 and container 2. This ping packet is travelling across the two CoreOS nodes and is encapsulated using VXLAN:

```
/ # ping -c1 10.1.1.2
PING 10.1.1.2 (10.1.1.2): 56 data bytes
64 bytes from 10.1.1.2: seq=0 ttl=62 time=0.786 ms

--- 10.1.1.2 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.786/0.786/0.786 ms
```

Flannel as a CNI plugin

As explained in *Chapter 1, CoreOS Overview*, APPC defines a Container specification that any Container runtime can use. For Container networking, APPC defines a **Container Network Interface** (**CNI**) specification. With CNI, the Container networking functionality can be implemented as a plugin. CNI expects plugins to support APIs with a set of parameters and the implementation is left to the plugin. The plugin implements APIs like adding a container to a network and removing container from the network with a defined parameter list.

This allows the implementation of network plugins by different vendors and also the reuse of plugins across different Container runtimes. The following figure shows the relationship between the **RKT** container runtime, **CNI** layer, and **Plugin like Flannel**. The **IPAM Plugin** is used to allocate an IP address to the containers and this is nested inside the initial networking plugin:



Setting up a three-node Vagrant CoreOS cluster with Flannel and Docker

The following example sets up a three-node Vagrant CoreOS cluster with the `etcd`, `fleet`, and `flannel` services turned on by default. In this example, `vxlan` is used for encapsulation. The following is the `cloud-config` used for this:

```
#cloud-config
coreos:
  etcd2:
    discovery: <update this>
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  fleet:
    public-ip: $public_ipv4
  flannel:
    interface: $public_ipv4
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
```

```
      command: start
    - name: flanneld.service
      drop-ins:
        - name: 50-network-config.conf
          content: |
            [Service]
            ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/
config '{ "Network": "10.1.0.0/16", "Backend": {"Type": "vxlan"}}'
      command: start
```

The following are the steps for this:

1. Clone the CoreOS Vagrant repository.
2. Change the instance count to three in `config.rb`.
3. Update the discovery token in the `cloud-config` user data.
4. Perform `vagrant up` to start the cluster.

For more details on the steps, refer to *Chapter 2*, *Setting up the CoreOS Lab*. We can test the container connectivity by starting `busybox` containers in both the hosts and checking that the ping is working between the two Containers.

Setting up a three-node CoreOS cluster with Flannel and RKT

Here, we will set up a three-node CoreOS cluster with RKT containers using the Flannel CNI networking plugin to set up the networking. This example will allow RKT containers across hosts to communicate with each other.

The following is the `cloud-config` used:

```
#cloud-config
coreos:
  etcd2:
    discovery: <update token>
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  fleet:
    public-ip: $public_ipv4
  flannel:
    interface: $public_ipv4
  units:
```

```
      - name: etcd2.service
        command: start
      - name: fleet.service
        command: start
      - name: flanneld.service
        drop-ins:
          - name: 50-network-config.conf
            content: |
              [Service]
              ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/
config '{ "network": "10.1.0.0/16" }'
      command: start
# Rkt configuration
write_files:
  - path: "/etc/rkt/net.d/10-containernet.conf"
    permissions: "0644"
    owner: "root"
    content: |
      {
        "name": "containernet",
        "type": "flannel"
      }
```

The `/etc/rkt/net.d/10-containernet.conf` file sets up the CNI plugin type as Flannel and RKT containers use this.

The following are the steps for this:

1. Clone the CoreOS Vagrant repository.
2. Change the instance count to three in `config.rb`.
3. Update the discovery token in the `cloud-config` user data.
4. Perform `vagrant up` to start the cluster.

Let's start a `busybox` container in `node1`:

```
core@core-01 ~ $ sudo rkt run --private-net --interactive --insecure-skip-verify docker://busybox
Downloading cfa753dfea5e: [==========================================] 667 KB/667 KB
Downloading d7057cb02084: [==========================================] 32 B/32 B
2015/10/03 15:22:54 Preparing stage1
2015/10/03 15:22:55 Loading image sha512-769c1c6b8a9f2576f7955df5c7fe20dc8327f8af73c6388820168644399bc0b
4
2015/10/03 15:22:55 Writing pod manifest
2015/10/03 15:22:55 Setting up stage1
2015/10/03 15:22:55 Writing image manifest
2015/10/03 15:22:55 Wrote filesystem to /var/lib/rkt/pods/run/979b4f9f-8617-4939-8b3d-e013a3861000
2015/10/03 15:22:55 Writing image manifest
2015/10/03 15:22:55 Pivoting to filesystem /var/lib/rkt/pods/run/979b4f9f-8617-4939-8b3d-e013a3861000
2015/10/03 15:22:55 Execing /init
```

The `ifconfig` output in `busybox node1` is as follows:

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 86:A1:95:1F:55:B9
          inet addr:10.1.26.2  Bcast:0.0.0.0  Mask:255.255.255.0
```

Start a `busybox` container in `node2`:

```
core@core-02 ~ $ sudo rkt run --private-net --interactive --insecure-skip-verify docker://busybox
Downloading cfa753dfea5e: [==========================================] 667 KB/667 KB
```

The `ifconfig` output in `busybox node2` is as follows:

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 7E:FB:B1:7B:1A:A6
          inet addr:10.1.67.2  Bcast:0.0.0.0  Mask:255.255.255.0
```

The following screenshot shows you the successful ping output across containers:

```
/ # ping -c1 10.1.67.2
PING 10.1.67.2 (10.1.67.2): 56 data bytes
64 bytes from 10.1.67.2: seq=0 ttl=60 time=0.670 ms
```

Note: `Docker.service` should not be started with RKT containers as the Docker bridge uses the same address that is allocated to Flannel for Docker container communication. Active work is going on to support running both Docker and RKT containers using Flannel. Some discussion on this topic can be found at `https://groups.google.com/forum/#!topic/coreos-user/Kl7ejtcRxbc`.

An AWS cluster using Flannel

Flannel can be used to provide Container networking between CoreOS nodes in the AWS cloud. In the following two examples, we will create a three-node CoreOS cluster in AWS using Flannel with VXLAN and Flannel with AWS VPC networking. These examples are based on the procedure described at `https://coreos.com/blog/introducing-flannel-0.5.0-with-aws-and-gce/`.

An AWS cluster using VXLAN networking

The following are the prerequisities for this:

1. Create a token for the three-node cluster from the discovery token service.
2. Set up a security group exposing the ports `ssh`, `icmp`, `2379`, `2380`, and `8472`. `8472` is used for VXLAN encapsulation.
3. Determine the AMI image ID using this link (`https://coreos.com/os/docs/latest/booting-on-ec2.html`) based on your AWS Zone, and update the channel based on your AWS zone and update channel. For the following example, we will use ami-150c1425, which is the latest 815 alpha image.

Create `cloud-config-flannel-vxlan.yaml` with the same content used for the Vagrant CoreOS cluster with Flannel and Docker, as specified in the previous section.

Use the following AWS CLI to start the three-node cluster:

```
aws ec2 run-instances --image-id ami-85ada4b5 --count 3 --instance-type
t2.micro --key-name "yourkey" --security-groups "coreos " --user-data
```

We can test connectivity across containers using two `busybox` containers in two CoreOS nodes as specified in the previous sections.

An AWS cluster using AWS-VPC

AWS VPC provides you with an option to create custom networking for the instances created in AWS. With AWS VPC, we can create subnets and route tables and configure custom IP addresses for the instances.

Flannel supports the encapsulation type, `aws-vpc`. When using this option, Flannel updates the VPC route table to route between instances by creating a custom route table per VPC based on the container IP addresses allocated to the individual node. From a data path perspective, there is no encapsulation such as UDP or VXLAN that's used. Instead, AWS VPC takes care of routing the packets to the appropriate instance using the route table configured by Flannel.

The following are the steps to create the cluster:

1. Create a discovery token for the three-node cluster.
2. Set up a security group exposing the ports `ssh`, `icmp`, `2379`, and `2380`.
3. Determine the AMI image ID using this link (`https://coreos.com/os/docs/latest/booting-on-ec2.html`). For the following example, we will use `ami-150c1425`, which is the latest 815 alpha image.

4. Create a VPC using the VPC wizard with a single public subnet. The following diagram shows you the VPC created from the AWS console:

```
smakam14@jungle1:~/coreos$ aws ec2 describe-vpcs --vpc-id vpc-410c4824
{
    "Vpcs": [
        {
            "VpcId": "vpc-410c4824",
            "InstanceTenancy": "default",
            "State": "available",
            "DhcpOptionsId": "dopt-b7f516d2",
            "CidrBlock": "10.0.0.0/16",
            "IsDefault": false
        }
    ]
}
```

5. Create an IAM policy, `demo-policy`, from the AWS console. This policy allows the instance to modify routing tables:

```
{
    "Version": "2012-10-17",
    "Statement": [
    {
            "Effect": "Allow",
            "Action": [
                "ec2:CreateRoute",
                "ec2:DeleteRoute",
                "ec2:ReplaceRoute"
            ],
            "Resource": [
                "*"
            ]
    },
    {
            "Effect": "Allow",
            "Action": [
                "ec2:DescribeRouteTables",
                "ec2:DescribeInstances"
            ],
            "Resource": "*"
    }
    ]
}
```

6. Create an IAM role, `demo-role`, and associate `demo-policy` created in the preceding code with this role.

7. Create `cloud-config-flannel-aws.yaml` with the following content. We will use the type as `aws-vpc`, as shown in the following code:

```
Cloud-config-flannel-aws.yaml:
#cloud-config
coreos:
  etcd2:
    discovery: <your token>
    advertise-client-urls: http://$private_
ipv4:2379,http://$private_ipv4:4001
    initial-advertise-peer-urls: http://$private_ipv4:2380
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
    - name: flanneld.service
      drop-ins:
        - name: 50-network-config.conf
          content: |
            [Service]
            ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/
config '{ "Network": "10.1.0.0/16" , "Backend": {"Type": "aws-
vpc"}}'
      command: start
```

Create a three-node CoreOS cluster with a security group, IAM role, `vpcid/subnetid`, `security group`, and `cloud-config` file as follows:

```
aws ec2 run-instances --image-id ami-150c1425 --subnet subnet-a58fc5c0
--associate-public-ip-address --iam-instance-profile Name=demo-role
--count 3 --security-group-ids sg-f22cb296 --instance-type t2.micro
--key-name "smakam-oregon"  --user-data file://cloud-config-flannel-aws.
yaml
```

Note: It is necessary to disable the source and destination checks to allow traffic from containers as the IP address for the containers is allocated by flannel and not by AWS. To do this, we need to go to each instance in the AWS console and select **Networking** | **change source/dest check** | **disable**.

Looking at the `etcdctl` output in one of the CoreOS nodes, we can see the following subnets allocated to each node of the three-node cluster.

```
/coreos.com/network/subnets/10.1.47.0-24
/coreos.com/network/subnets/10.1.12.0-24
/coreos.com/network/subnets/10.1.14.0-24
```

Flannel will go ahead and update the VPC route table to route the preceding subnets based on the instance ID on which the subnets are present. If we check the VPC route table, we can see the following routes, which match the networks created by Flannel:

Destination	Target	Status	Propagated
10.0.0.0/16	local	Active	No
0.0.0.0/0	igw-3d0d8058	Active	No
10.1.12.0/24	eni-32356954 / i-8cfe1948	Active	No
10.1.14.0/24	eni-35356953 / i-8ffe194b	Active	No
10.1.47.0/24	eni-33356955 / i-8dfe1949	Active	No

At this point, we can test connectivity across containers using two `busybox` containers in two CoreOS nodes, as specified in the previous sections.

A GCE cluster using Flannel

Flannel can be used to provide Container networking between CoreOS nodes in the GCE cloud. In the following two examples, we will create a three-node CoreOS cluster using Flannel with VXLAN and Flannel with GCE networking. These examples are based on the procedure described at `https://coreos.com/blog/introducing-flannel-0.5.0-with-aws-and-gce/`.

GCE cluster using VXLAN networking

The following are the prerequisities for this:

1. Create a token for the three-node cluster from the discovery token service.
2. Set up a security group exposing the ports `ssh`, `icmp`, `2379`, `2380`, and `8472`. `8472` is used for VXLAN encapsulation.
3. Determine the AMI image ID using this link (`https://coreos.com/os/docs/latest/booting-on-google-compute-engine.html`). We will use alpha image 815 for the following example.

Create `cloud-config-flannel-vxlan.yaml` with the same content that was used for the Vagrant CoreOS cluster with Flannel and Docker specified in the previous section.

The following command can be used to set up a three-node CoreOS cluster in GCE with Flannel and VXLAN encapsulation:

```
gcloud compute instances create core1 core2 core3 --image https://www.
googleapis.com/compute/v1/projects/coreos-cloud/global/images/coreos-
alpha-815-0-0-v20150924 --zone us-central1-a --machine-type n1-standard-1
--tags coreos --metadata-from-file user-data=cloud-config-flannel-vxlan.
yaml
```

A ping test across containers in different hosts can be done to verify that the Flannel control and data path is working fine.

A GCE cluster using GCE networking

Similar to AWS VPC, the Google cloud also has its cloud networking service that provides you with the capability to create custom subnets, routes, and IP addresses.

The following are the steps to create a three-node CoreOS cluster using flannel and GCE networking:

1. Create a token for the three-node cluster from the discovery token service.
2. Create a custom network, `customnet`, with firewall rules allowing TCP ports `2379` and `2380`. The following is the custom network that I created with subnet `10.10.0.0/16`:

```
smakam14@jungle1:~/coreos$ gcloud compute networks list
NAME      IPV4_RANGE    GATEWAY_IPV4
customnet 10.10.0.0/16  10.10.0.1
```

3. Create `cloud-config-flannel-gce.yaml` with the following content. Use the Flannel type as `gce`:

```
#cloud-config
coreos:
  etcd2:
    discovery: <yourtoken>
    advertise-client-urls: http://$private_
ipv4:2379,http://$private_ipv4:4001
    initial-advertise-peer-urls: http://$private_ipv4:2380
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380
  units:
```

```
      - name: etcd2.service
        command: start
      - name: fleet.service
        command: start
      - name: flanneld.service
        drop-ins:
          - name: 50-network-config.conf
            content: |
              [Service]
              ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/
config '{ "Network": "10.1.0.0/16" , "Backend": {"Type": "gce"}}'
        command: start
```

4. Create three CoreOS instances with a `customnet` network, IP forwarding turned on, and scope for the instance to modify the route table:

   `gcloud compute instances create core1 core2 core3 --image https://www.googleapis.com/compute/v1/projects/coreos-cloud/global/images/coreos-alpha-815-0-0-v20150924 --zone us-central1-a --machine-type n1-standard-1 --network customnet --can-ip-forward --scopes compute-rw --metadata-from-file user-data=cloud-config-flannel-gce.yaml`

The following are the Flannel networks for containers created by each node:

```
/coreos.com/network/subnets/10.1.5.0-24
/coreos.com/network/subnets/10.1.73.0-24
/coreos.com/network/subnets/10.1.38.0-24
```

Let's look at the routing table in GCE. As shown by the following output, Flannel has updated the GCE route table for the container networks:

```
smakam14@jungle1:~/coreos$ gcloud compute routes list --sort-by NETWORK
NAME                            NETWORK   DEST_RANGE    NEXT_HOP                      PRIORITY
default-route-66230db66eaf4fac customnet 10.10.0.0/16
default-route-8ea9d62be0ff0845 customnet 0.0.0.0/0     default-internet-gateway      1000
flannel-10-1-38-0-24           customnet 10.1.38.0/24  us-central1-a/instances/core1 1000
flannel-10-1-5-0-24            customnet 10.1.5.0/24   us-central1-a/instances/core2 1000
flannel-10-1-73-0-24           customnet 10.1.73.0/24  us-central1-a/instances/core3 1000
```

At this point, we should have Container connectivity across nodes.

Experimental multitenant networking

By default, Flannel creates a single network, and all the nodes can communicate with each other over the single network. This poses a security risk when there are multiple tenants using the same network. One approach to achieve multitenant networking is using multiple instances of flannel managing each tenant. This can get cumbersome to set up. As of version 0.5.3, Flannel has introduced multinetworking in the experimental mode, where a single Flannel daemon can manage multinetworks with isolation. When there are multiple tenants using the cluster, a multinetwork mode would help in isolating each tenant's traffic.

The following are the steps for this:

1. Create subnet configurations for multiple tenants. This can be done by reserving a subnet pool in `etcd`. The following example sets up three networks, `blue`, `green`, and `red`, each having a different subnet:

```
etcdctl set /coreos.com/network/blue/config  '{ "Network":
"10.1.0.0/16", "Backend": { "Type": "vxlan", "VNI": 1 } }'

etcdctl set /coreos.com/network/green/config '{ "Network":
"10.2.0.0/16", "Backend": { "Type": "vxlan", "VNI": 2 } }'

etcdctl set /coreos.com/network/red/config   '{ "Network":
"10.3.0.0/16", "Backend": { "Type": "vxlan", "VNI": 3 } }'
```

2. Start the Flannel agent with the networks that this Flannel agent needs to be part of. This will take care of reserving the IP pool per node per network. In this example, we have started the flannel agent to be part of all three networks, `blue`, `green`, and `red`:

```
sudo flanneld --networks=blue,green,red &
```

Flannel picked three subnet ranges for the three networks, as shown in the following screenshot. `10.1.87.0/24` is allocated for the `blue` network, `10.2.4.0/24` is allocated for the `green` network, and `10.3.93.0/24` is allocated for the `red` network:

```
/coreos.com/network
/coreos.com/network/blue
/coreos.com/network/blue/config
/coreos.com/network/blue/subnets
/coreos.com/network/blue/subnets/10.1.87.0-24
/coreos.com/network/green
/coreos.com/network/green/subnets
/coreos.com/network/green/subnets/10.2.4.0-24
/coreos.com/network/green/config
/coreos.com/network/red
/coreos.com/network/red/config
/coreos.com/network/red/subnets
/coreos.com/network/red/subnets/10.3.93.0-24
```

Under `/run/flannel`, multiple networks can be seen, as follows:

```
core@core-01 /run/flannel/networks $ ls
blue.env  green.env  red.env
core@core-01 /run/flannel/networks $ cat blue.env
FLANNEL_NETWORK=10.1.0.0/16
FLANNEL_SUBNET=10.1.87.1/24
FLANNEL_MTU=1450
FLANNEL_IPMASQ=false
```

Now, we can start the Docker or Rkt container with the appropriate tenant network created. At this point, there is no automatic integration of `flanneld.service` with multinetworks; this has to be done manually.

The following link is a related Google discussion on this topic:

`https://groups.google.com/forum/#!topic/coreos-user/EIF-yGNWkL4`

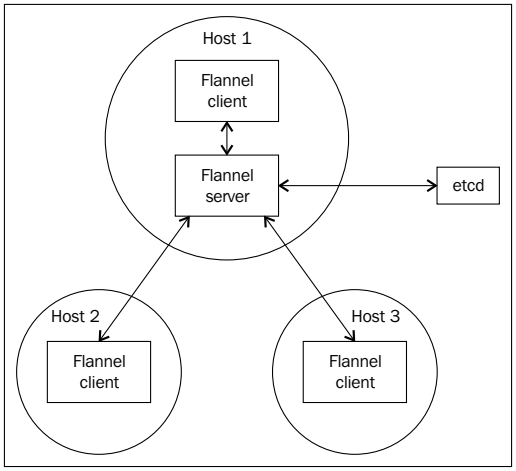
Experimental client-server networking

In the default Flannel mode, there is a flannel agent in each node, and the backend data is maintained in `etcd`. This keeps Flannel stateless. In this mode, there is a requirement for each Flannel node to run `etcd`. Flannel client-server mode is useful in the following scenarios:

- Only the master node runs `etcd` and Worker nodes need not run `etcd`. This is useful from both the performance and security perspectives.
- When using other backends such as AWS with flannel, it's necessary to store the AWS key, and when using the client-server model, the key can be present in the master node only; this is again important from a security perspective.

Flannel client-server feature is currently in experimental mode as of Flannel version 0.5.3.

The following figure describes the interconnection between different components for the Flannel client-server networking:



If necessary, we can use secure (HTTPS) means of communication both from the flanneld server to the etcd as well as between the flanneld client and server.

Setting up client-server Flannel networking

Let's start with a three-node CoreOS cluster without Flannel running on any node. Start the `flanneld` server and client in `node1` and client in `node2` and `node3`.

Start flannel server as shown in the following screenshot:

```
core@core-01 ~ $ flanneld --listen=0.0.0.0:8888 &
[1] 934
core@core-01 ~ $ I1006 14:28:26.800166 00934 main.go:111] Installing signal handlers
I1006 14:28:26.800280 00934 main.go:124] running as server
```

Start flannel client as shown in the following screenshot:

It is necessary to specify the interface with `eth1` as an argument as `eth0` is used as the NAT interface and is common across all nodes, eth1 is unique across nodes:

```
core@core-01 ~ $ sudo flanneld --remote=172.17.8.101:8888 --iface=eth1 &
[2] 1105
core@core-01 ~ $ I1006 14:37:41.994762 01105 main.go:111] Installing signal handlers
I1006 14:37:41.995532 01105 manager.go:106] Using 172.17.8.101 as external interface
I1006 14:37:41.995778 01105 manager.go:107] Using 172.17.8.101 as external endpoint
I1006 14:37:42.003977 00934 http_logger.go:47] GET /v1/_/config - 200
I1006 14:37:42.006247 00934 etcd.go:212] Picking subnet in range 10.1.1.0 ... 10.1.255.0
I1006 14:37:42.041716 00934 etcd.go:92] Subnet lease acquired: 10.1.36.0/24
```

After starting the client in `node2` and `node3`, let's look at the `etcd` output in `node1` showing the three subnets acquired by three CoreOS nodes:

```
/coreos.com/network/subnets/10.1.36.0-24
/coreos.com/network/subnets/10.1.69.0-24
/coreos.com/network/subnets/10.1.22.0-24
```

To start `docker.service` manually, we first need to create `flannel_docker_opts.env` as follows:

```
/usr/bin/docker run --net=host --rm -v /run:/run \
  quay.io/coreos/flannel:0.5.3 \
  /opt/bin/mk-docker-opts.sh -d /run/flannel_docker_opts.env -i
```

The following image is the created `flannel_docker_opts.env`:

```
core@core-01 ~ $ cat /run/flannel_docker_opts.env
DOCKER_OPT_BIP="--bip=10.1.36.1/24"
DOCKER_OPT_IPMASQ="--ip-masq=true"
DOCKER_OPT_MTU="--mtu=1472"
```

Now, we can start `docker.service`, which uses environment variables in `flannel_docker_opts.env`.

Start `docker.service`:

```
sudo systemctl start docker.service
```

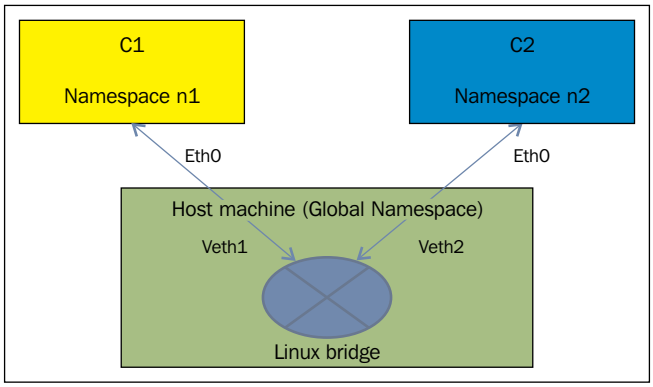
As we can see, the docker bridge gets the IP address in the range allocated to this node:

```
docker0: flags=4099<UP,BROADCAST,MULTICAST>  mtu 1500
        inet 10.1.36.1  netmask 255.255.255.0  broadcast 0.0.0.0
```

This feature is currently experimental. There are plans to add a server failover feature in future.

Docker networking

The following is the Docker networking model to interconnect containers in a single host:



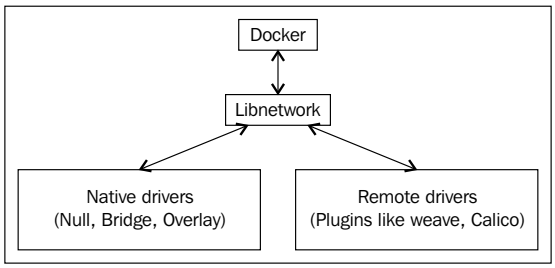
Each Container resides in its own networking namespace and uses a Linux bridge on the host machine to talk to each other. More details on Docker networking options can be found at `https://docs.docker.com/engine/userguide/networking/dockernetworks/`. The following are the networking options available as of Docker release 1.9:

- `--net=bridge`: This is the default option that Docker provides, where containers connect to the Linux `docker` bridge using a veth pair.
- `--net=host`: In this option, there is no new network namespace created for the container, and the container shares the same network namespace as the host machine.
- `--net= (the container name or ID)`: In this option, the new container shares the same network namespace as the specified container in the `net` option. (For example: `sudo docker run -ti –name=ubuntu2 –net=container:ubuntu1 ubuntu:14.04 /bin/bash`. Here, the `ubuntu2` container shares the same network namespace as the `ubuntu1` container.)
- `--net=none`: In this option, the container does not get allocated a new network namespace. Only the loopback interface is created in this case. This option is useful in scenarios where we want to create our own networking options for the container or where there is no need for connectivity.
- `--net=overlay`: This option was added in Docker release 1.9 to support overlay networking that allows Containers across hosts to be able to talk to each other.

Docker experimental networking

As of Docker release 1.8, Docker did not have a native solution to connect Containers across hosts. With the Docker experimental release, we can connect Containers across hosts using the Docker native solution as well as external networking plugins to connect Containers across hosts.

The following figure illustrates this:



The following are some notes on the Docker libnetwork solution:

- Docker runtime was previously integrated with the networking module and there was no way to separate them. Libnetwork is the new networking library that provides the networking functionality and is separated from Core Docker. Docker 1.7 release has already included the libnetwork and is backward-compatible from the end user's perspective.
- Drivers implement the APIs provided by libnetwork. Docker is leaning towards a plugin approach for major functionalities such as Networking, Storage, and Orchestration where Docker provides a native solution that can be substituted with technologies from other vendors as long as they implement the APIs provided by the common library. In this case, Bridge and Overlay are the native Docker networking drivers and remote drivers can be implemented by a third party. There are already many remote drivers available, such as Weave and Calico.

Docker experimental networking has the following concepts:

- The Docker container attaches to the network using the endpoint or service.
- Multiple endpoints share a network. In other words, only endpoints located in the same network can talk to each other.
- When creating the network, the network driver can be mentioned. This can be a Docker-provided driver, such as Overlay, or an external driver, such as Weave and Calico.

- Libnetwork provides service discovery, where Containers can discover other endpoints in the same network. There is a plan in the future to make service discovery a plugin. Services can talk to each other using the service name rather than the IP address. Currently, Consul is used for service discovery; this might change later.
- Shared storage such as `etcd` or `consul` is used to determine the nodes that are part of the same cluster.

A multinetwork use case

With the latest Docker networking enhancements, Containers can be part of multiple networks and only Containers in the same network can talk to each other. To illustrate these concepts, let's take a look at the following example:

1. Set up two nginx containers and one HAProxy Container in the backend network, `be`.
2. Add the HAProxy Container in the frontend network, `fe`, as well.
3. Connect to the HAProxy Container using the busybox Container in the frontend network, `fe`. As the busybox Container is in the `fe` network and nginx Containers are in the `be` network, they cannot talk to each other directly.
4. The Haproxy Container will load balance the web connection between the two nginx backend Containers.

The following are the command details:

Create the `fe` and `be` networks:

```
docker network create be
docker network create fe
```

Create two nginx containers in the `be` network:

```
docker run --name nginx1 --net be -v ~/haproxy/nginx1.html:/usr/share/
nginx/html/index.html -d nginx
docker run --name nginx2 --net be -v ~/haproxy/nginx2.html:/usr/share/
nginx/html/index.html -d nginx
```

Create `haproxy` in the `be` network:

```
docker run -d --name haproxy --net be -v ~/haproxy/haproxy.cfg:/usr/
local/etc/haproxy/haproxy.cfg haproxy
```

Attach `haproxy` to the `fe` network:

```
docker network connect fe haproxy
```

Create a busybox container in the `fe` network accessing the `haproxy` web page:

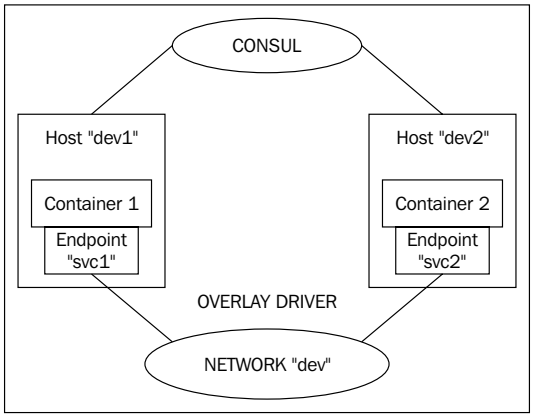
```
docker run -it --rm --net fe busybox wget -qO- haproxy/index.html
```

If we try running the busybox container multiple times, it will switch between `nginx1` and `nginx2` web server outputs.

The Docker overlay driver

The following example shows you how to do multihost container connectivity using the Docker experimental overlay driver. I have used a Ubuntu VM for the following example and not CoreOS because the experimental docker overlay driver needs a new kernel release, which is not yet available in CoreOS.

The following figure illustrates the use case that is being tried in this example:



The following is a summary of the steps:

- Create two hosts with experimental Docker installed.
- Install Consul on both the hosts with one of the hosts acting as the consul server. Consul is needed to store common data that is used for inter-container communication.
- Start Docker with Consul as the key store mechanism on both hosts.
- Create containers with different endpoints on both hosts sharing the same network.

The first step is to create 2 host machines with experimental Docker installed.

The following set of commands creates two Docker hosts using docker-machine. We have used docker-machine with a custom ISO image for experimental Docker:

```
docker-machine create -d virtualbox --virtualbox-boot2docker-url=http://
sirile.github.io/files/boot2docker-1.9.iso dev1
docker-machine create -d virtualbox --virtualbox-boot2docker-url=http://
sirile.github.io/files/boot2docker-1.9.iso dev2
```

Install Consul on both nodes. The following command shows you how to download and install consul:

```
curl -OL https://dl.bintray.com/mitchellh/consul/0.5.2_linux_amd64.zip
unzip 0.5.2_linux_amd64.zip
sudo mv consul /usr/local/bin/
```

Start the consul server and docker daemon with the consul keystore in `node1`:

The following set of commands starts the consul server and Docker daemon with the consul agent in `node1`:

```
Docker-machine ssh dev1

consul agent -server -bootstrap -data-dir /tmp/consul
-bind=192.168.99.100 &

sudo docker -d --kv-store=consul:localhost:8500 --label=com.docker.
network.driver.overlay.bind_interface=eth1
```

Start the consul agent and Docker daemon with the consul keystore in `node2`:

The following set of commands starts the consul agent and Docker daemon with the consul agent in `node2`:

```
Docker-machine ssh dev2

consul agent -data-dir /tmp/consul -bind 192.168.99.101 &

consul join 192.168.99.100 &

sudo docker -d --kv-store=consul:localhost:8500 --label=com.docker.
network.driver.overlay.bind_interface=eth1 --label=com.docker.network.
driver.overlay.neighbor_ip=192.168.99.100
```

Start the container with the `svc1` service, `dev` network, and `overlay` driver in `node1`:

```
docker run -i -t --publish-service=svc1.dev.overlay busybox
```

Start the container with the `svc2` service, `dev` network, and `overlay` driver in node2:

```
docker run -i -t --publish-service=svc2.dev.overlay busybox
```

As we can see, we are able to ping `svc1` and `svc2` from `node1` successfully:

```
/ # ping -c1 svc1.dev
PING svc1.dev (172.21.0.1): 56 data bytes
64 bytes from 172.21.0.1: seq=0 ttl=64 time=0.045 ms

--- svc1.dev ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.045/0.045/0.045 ms
/ # ping -c1 svc2.dev
PING svc2.dev (172.21.0.2): 56 data bytes
64 bytes from 172.21.0.2: seq=0 ttl=64 time=17.607 ms

--- svc2.dev ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 17.607/17.607/17.607 ms
```

Note: The overlay driver needs the Linux kernel version 3.16 or higher.

The external networking calico plugin

In this example, we will illustrate how to do Container networking using Calico as a plugin to the Docker libnetwork. This support was available originally in experimental networking and later in the Docker 1.9 release. More details about the Calico networking approach are mentioned in the following Calico networking section. This example is based on `https://github.com/projectcalico/calico-containers/blob/master/docs/calico-with-docker/docker-network-plugin/README.md`. To set up a CoreOS Vagrant cluster for Calico, we can use the procedure in `https://github.com/projectcalico/calico-containers/blob/master/docs/calico-with-docker/VagrantCoreOS.md`.

After setting up a Vagrant CoreOS cluster, we can see the two nodes of the CoreOS cluster. We should make sure that `etcd` is running successfully, as shown in the following output:

```
core@calico-01 ~ $ etcdctl member list
ce2a822cea30bfca: name=d4a09514d73b44e595d0fa6051633376c peerURLs=http://localhost:2380,http://localhost:7001 clientURLs=http://172.17.8.101:2
379,http://172.17.8.101:4001
```

The following are the steps to get Calico working with Docker as a networking plugin:

Start Calico in both nodes with the libnetwork option:

```
sudo calicoctl node --libnetwork
```

We should see the following Docker containers in both nodes:

```
core@calico-01 ~ $ docker ps
CONTAINER ID        IMAGE                           COMMAND             CREATED             STATUS              PORTS               NAMES
a54eebe5614b        calico/node-libnetwork:latest   "./start.sh"        20 minutes ago      Up 20 minutes                           calico-
libnetwork
742cfb3b4230        calico/node:latest              "/sbin/start_runit"   20 minutes ago      Up 20 minutes                           calico-
node
```

Create the `net1` network with Calico driver:

```
docker network create --driver=calico --subnet=192.168.0.0/24 net1
```

This gets replicated to all the nodes in the cluster. The following is the network list in `node2`:

```
core@calico-02 ~ $ docker network ls
NETWORK ID          NAME                DRIVER
e01fa1bb7d4b        net1                calico
7bd5ae53ebfa        docker_gwbridge     bridge
65ffd3e52b5a        none                null
be212eb06072        host                host
c168818e2a8a        bridge              bridge
```

- Create container 1 in `node1` with the `net1` network:

  ```
  docker run --net net1 --name workload-A -tid busybox
  ```

- Create container 2 in `node2` with the `net1` network:

  ```
  docker run --net net1 --name workload-B -tid busybox
  ```

Now, we can ping the two containers as follows:

```
core@calico-01 ~ $ docker exec workload-A ping -c 1 workload-B
PING workload-B (192.168.0.3): 56 data bytes
64 bytes from 192.168.0.3: seq=0 ttl=62 time=0.898 ms

--- workload-B ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.898/0.898/0.898 ms
```

The Docker 1.9 update

Docker 1.9 got released at the end of October 2015 that transitioned the experimental networking into production. There could be minor modifications necessary to the Docker networking examples in this chapter, which were tried with the Docker 1.8 experimental networking version.

With Docker 1.9, multihost networking is integrated with Docker Swarm and Compose. This allows us to orchestrate a multicontainer application spread between multiple hosts with a single command and the multi-host Container networking will be handled automatically.

Other Container networking technologies

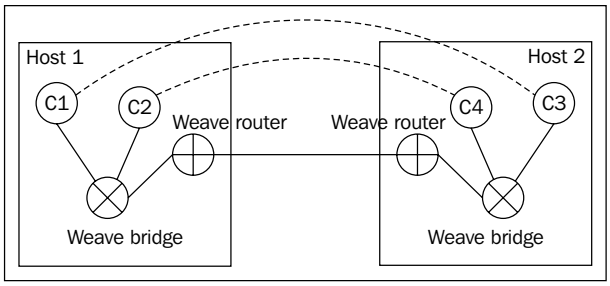
Weave and Calico are open source projects, and they develop Container networking technologies for Docker. Kubernetes is a Container orchestration open source project and it has specific networking requirements and implementations for Containers. There are also other projects such as Cisco Contiv (`https://github.com/contiv/netplugin`) that is targeted at Container networking. Container technologies like Weave, Calico and Contiv have plans to integrate with Rkt Container runtime in the future.

Weave networking

Weaveworks has developed a solution to provide Container networking. The following are some details of their solution:

- Weave creates a Weave bridge as well as a Weave router in the host machine.
- The Weave router establishes both TCP and UDP connections across hosts to other Weave routers. A TCP connection is used for discovery- and protocol-related exchange. UDP is used for data encapsulation. Encryption can be done if necessary.
- The Weave bridge is configured to sniff the packets that need to be sent across hosts and redirected to the Weave router. For local switching, the Weave router is not used.
- Weave's Weavenet product provides you with container connectivity. They also have Weavescope that provides container visibility and Weaverun that provides service discovery and load balancing.
- Weave is also available as a Docker plugin integrated with the Docker release 1.9.

The following figure illustrates the solution from Weave:



To run Weave on CoreOS, I used cloud-config from `https://github.com/lukebond/coreos-vagrant-weave`. In the following example, we will create containers in two CoreOS nodes and use Weave to communicate with each other. In this example, we have not used the Docker Weave plugin but used environment variables to communicate between Docker and Weave.

The following are the steps to create a Weave cluster:

1. Clone the repository (`git clone https://github.com/lukebond/coreos-vagrant-weave.git`).
2. Change the number of instances in `config.rb` to `3`.
3. Get a new discovery token for node count 3 and update it in the user data.
4. Perform `vagrant up` to start the cluster.

The `cloud-config` file takes care of downloading Weave agents in each node and starting them.

The following section of the service file downloads the Weave container:

```
ExecStartPre=/usr/bin/wget -N -P /opt/bin \
    https://raw.github.com/zettio/weave/master/weave
ExecStartPre=/usr/bin/chmod +x /opt/bin/weave
ExecStartPre=/usr/bin/docker pull zettio/weave:latest
```

The following section of the service file starts the Weave container:

```
[Service]
EnvironmentFile=/etc/weave.%H.env
ExecStartPre=/opt/bin/weave launch $WEAVE_LAUNCH_ARGS
ExecStart=/usr/bin/docker logs -f weave
```

On each of the nodes, we can see the following Weave containers started:

```
core@core-01 ~ $ docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS              PORTS
                                                                            NAMES
89baf8dc45ca        weaveworks/weaveexec:latest    "/home/weave/weavepro"   52 seconds ago      Up 51 seconds
                                                                            weaveproxy
6e0ac9c5dd72        weaveworks/weave:latest        "/home/weave/weaver -"   53 seconds ago      Up 52 seconds       172.18.42.1:53->53/tcp,
 172.18.42.1:53->53/udp, 0.0.0.0:6783->6783/tcp, 0.0.0.0:6783->6783/udp   weave
```

Before starting application containers, we need to set the environment variables so that Weave can intercept Docker commands and create their own networking. As part of starting Weave in `Weave.service`, environment variables have already been set up. The following command in the node shows this:

```
core@core-01 ~ $ weave env
export DOCKER_HOST=unix:///var/run/weave/weave.sock ORIG_DOCKER_HOST=
```

Source the Weave environment as follows:

```
core@core-01 ~ $ eval $(weave env)
```

Let's start busybox containers in two CoreOS nodes:

```
core@core-01 ~ $ docker run -ti busybox sh
```

Let's look at the Weave interface created in the busybox container of CoreOS node1:

```
ethwe     Link encap:Ethernet  HWaddr 7A:B0:3E:70:F8:0E
          inet addr:10.32.0.1  Bcast:0.0.0.0  Mask:255.240.0.0
```

Let's look at the Weave interface created in the busybox container of CoreOS node2:

```
ethwe     Link encap:Ethernet  HWaddr 36:8E:27:F9:DC:41
          inet addr:10.40.0.0  Bcast:0.0.0.0  Mask:255.240.0.0
```

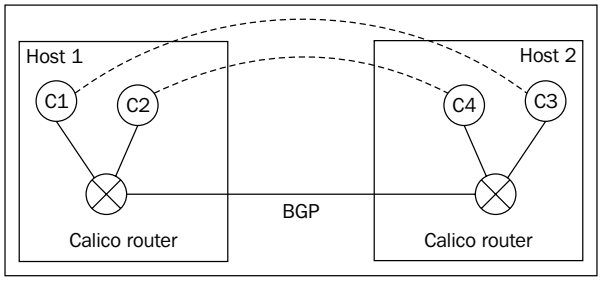
Now, we can successfully ping between the two containers. As part of Docker 1.9, Weave is available as a Docker networking plugin and this makes configuration much easier.

Calico networking

Calico provides you with a Container networking solution for Docker similar to Weave. The following are some details of Calico's implementation:

- Calico provides container networking directly at L3 without using overlay technologies
- Calico uses BGP for route distribution
- There are two components of Calico: BIRD, which is used for route distribution and FELIX, which is an agent in each node that does discovery and routing
- Calico is also available as a Docker networking plugin integrated with Docker release 1.9

The following figure illustrates the Calico data path:



Setting up Calico with CoreOS

I followed the procedure at `https://github.com/projectcalico/calico-containers/blob/master/docs/calico-with-docker/VagrantCoreOS.md` to set up a two-node CoreOS cluster.

The first step is checking out the repository:

```
git clone https://github.com/projectcalico/calico-docker.git
```

There are three approaches described by Calico for Docker networking:

- Powerstrip: Calico uses an HTTP proxy to listen to Docker calls and set up networking.
- Default networking: Docker Containers are set up with no networking. Using Calico, network endpoints are added and networking is set up.
- Libnetwork: Calico is integrated with Docker libnetwork as of Docker release 1.9. This will be the long-term solution.

In the following example, we have used the default networking approach to set up Container connectivity using Calico.

The following are the steps needed to set up the default networking option with Calico:

1. Start calicoctl in all the nodes.
2. Start the containers with the `--no-net` option.
3. Attach the calico network specifying the IP address to each container.
4. Create a policy profile. Profiles set up the policy that allows containers to talk to each other.
5. Attach profiles to the container.

The following commands set up a container in `node1` and `node2` and establish a policy that allows containers to talk to each other.

Execute the following commands on `node1`:

```
docker run --net=none --name workload-A -tid busybox
sudo calicoctl container add workload-A 192.168.0.1
calicoctl profile add PROF_A_B
calicoctl container workload-A profile append PROF_A_B
```

This starts the docker container, attaches the calico endpoint, and applies the profile to allow Container connectivity.

Execute the following commands on `node2`:

```
docker run --net=none --name workload-B -tid busybox
sudo calicoctl container add workload-B 192.168.0.2
calicoctl container workload-B profile append PROF_A_B
```

This starts the docker container, attaches the calico endpoint, and applies the same profile as in the preceding commands to allow Container connectivity.

Now, we can test intercontainer connectivity:

```
core@calico-02 ~ $ docker exec workload-B ping -c 1 192.168.0.1
PING 192.168.0.1 (192.168.0.1): 56 data bytes
64 bytes from 192.168.0.1: seq=0 ttl=62 time=0.839 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.839/0.839/0.839 ms
```

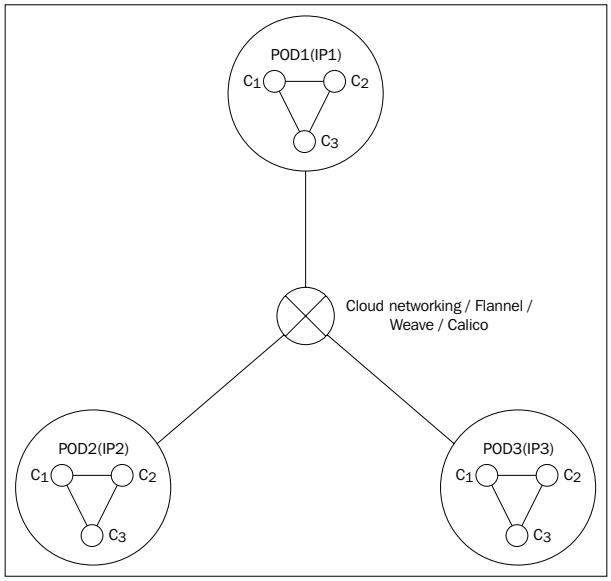
Kubernetes networking

Kubernetes is a Container orchestration service. Kubernetes is an open source project that's primarily driven by Google. We will discuss about Kubernetes in the next chapter on Container orchestration. In this chapter, we will cover some of the Kubernetes networking basics.

The following are some details as to how Kubernetes does the networking of containers:

- Kubernetes has a concept called Pods, which is a collection of closely-tied containers. For example, a service and its logging service can be a single pod. Another example of a pod could be a service and sidekick service that checks the health of the main service. A single pod with its associated containers is always scheduled on one machine.
- Each pod gets an IP address. All containers within a pod share the same IP address.
- Containers within a pod share the same network namespace. For containers within a pod to communicate, they can use a regular process-based communication.
- Pods can communicate with each other using a cloud networking VPC-based approach or container networking solution such as Flannel, Weave, or Calico.
- As pods are ephemeral, Kubernetes has a unit called service. Each service has an associated virtual IP address and proxy agent running on the nodes' load balancers and directs traffic to the right pod.

The following is an illustration of how Pods and Containers are related and how they communicate:



Summary

In this chapter, we covered different Container networking technologies with a focus on Container networking in CoreOS. There are many companies trying to solve this Container networking problem. CNI and Flannel have become the default for CoreOS and Libnetwork has become the default for Docker. Having standards and pluggable networking architecture is good for the industry as this allows interoperability. Container networking is still in the early stages, and it will take some time for the technologies to mature in this area. In the next chapter, we will discuss about CoreOS storage management.

References

- Flannel docs: `https://coreos.com/flannel/docs/latest/`
- Flannel GitHub page: `https://github.com/coreos/flannel`
- CNI spec: `https://github.com/appc/cni/blob/master/SPEC.md`
- Flannel with AWS and GCE: `https://coreos.com/blog/introducing-flannel-0.5.0-with-aws-and-gce/`
- Weaveworks: `https://github.com/weaveworks/weave`
- Libnetwork: `https://github.com/docker/libnetwork`
- Docker experimental: `https://github.com/docker/docker/tree/master/experimental`
- Calico: `https://github.com/projectcalico/calico-docker`
- Kubernetes: `http://kubernetes.io/`

Further reading and tutorials

- The Flannel CoreOS Fest presentation: `https://www.youtube.com/watch?v=_HYeSaGtEYw`
- The Calico and Weave presentation: `https://giantswarm.io/events/2015-04-20-docker-coreos/`
- Contiv netplugin: `https://github.com/contiv/netplugin`
- Kubernetes networking: `https://github.com/kubernetes/kubernetes/blob/release-1.1/docs/admin/networking.md`

6
CoreOS Storage Management

Storage is a critical component of distributed infrastructure. The initial focus of Container technology was on Stateless Containers with Storage managed by traditional technologies such as NAS and SAN. Stateless Containers are typically web applications such as NGINX and Node.js where there is no need to persist data. In recent times, there has been a focus on Stateful Containers and there are many new technologies being developed to achieve Stateful Containers. Stateful Containers are databases such as SQL and redis that need data to be persisted. CoreOS and Docker integrates well with different Storage technologies and there is active work going on to fill the gaps in this area.

Following three aspects of CoreOS storage will be covered in this chapter:

- The CoreOS base filesystem and partition table
- The Container filesystem, which is composed of the Union filesystem and **Copy-on-write** (**CoW**) storage driver
- The Container data volumes for shared data persistence, which can be local, distributed, or shared external storage

The following topics will be covered in this chapter:

- The CoreOS filesystem and mounting AWS EBS and NFS storage to the CoreOS filesystem
- The Docker Container filesystem for storing Container images which includes both storage drivers and the Union filesystem.

- Docker data volumes
- Container data persistence using Flocker, GlusterFS and Ceph

Storage concepts

The following are some storage terms along with their basic definitions that we will use in this chapter and beyond:

- **Local storage**: This is Storage attached to the localhost. An example is a local hard disk with ZFS.
- **Network storage**: This is a common storage accessed through a network. This can either be SAN or a cluster storage such as Ceph and GlusterFS.
- **Cloud storage**: This is Storage provided by a cloud provider such as AWS EBS, OpenStack Cinder, and Google cloud storage.
- **Block storage**: This requires low latency and is typically used for an OS-related filesystem. Some examples are AWS EBS and OpenStack Cinder.
- **Object storage**: This is used for immutable storage items where latency is not a big concern. Some examples are AWS S3 and OpenStack Swift.
- **NFS**: This is a distributed filesystem. This can be run on top of any cluster storage.

The CoreOS filesystem

We covered the details of the CoreOS partition table in *Chapter 3*, *CoreOS Autoupdate*. The following screenshot shows the default partitioning in the AWS CoreOS cluster:

```
core@ip-172-31-46-220 ~ $ df -k
Filesystem     1K-blocks   Used Available Use% Mounted on
devtmpfs          497168      0    497168   0% /dev
tmpfs             510100      0    510100   0% /dev/shm
tmpfs             510100    260    509840   1% /run
tmpfs             510100      0    510100   0% /sys/fs/cgroup
/dev/xvda9       5706380  19972   5410080   1% /
/dev/xvda3       1007760 354076    601668  38% /usr
tmpfs             510100      0    510100   0% /media
tmpfs             510100      0    510100   0% /tmp
/dev/xvda6        110576     60    101344   1% /usr/share/oem
```

By default, CoreOS uses root partitioning for the Container filesystem. In the preceding table, `/dev/xvda9` will be used to store Container images.

Following output shows Docker using Ext4 filesystem with Overlay storage driver in a CoreOS node running in AWS:

```
core@ip-172-31-46-220 ~ $ docker info
Containers: 0
Images: 0
Storage Driver: overlay
 Backing Filesystem: extfs
```

To get extra storage, external storage can be mounted in CoreOS.

Mounting the AWS EBS volume

Amazon **Elastic Block Store** (**EBS**) provides you with persistent block-level storage volumes to be used with Amazon EC2 instances in the AWS cloud. The following example shows you how to add an extra EBS volume to the CoreOS node running in AWS and use it for the Container filesystem.

Rename the following `cloud-config` as `cloud-config-mntdocker.yml`:

```
#cloud-config
coreos:
  etcd2:
    name: etcdserver
    initial-cluster: etcdserver=http://$private_ipv4:2380
    advertise-client-urls: http://$private_ipv4:2379,http://$private_
ipv4:4001
    initial-advertise-peer-urls: http://$private_ipv4:2380
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
    - name: format-ephemeral.service
      command: start
      content: |
        [Unit]
        Description=Formats the ephemeral drive
        After=dev-xvdf.device
        Requires=dev-xvdf.device
        [Service]
        Type=oneshot
        RemainAfterExit=yes
```

```
      ExecStart=/usr/sbin/wipefs -f /dev/xvdf
      ExecStart=/usr/sbin/mkfs.btrfs -f /dev/xvdf
  - name: var-lib-docker.mount
    command: start
    content: |
      [Unit]
      Description=Mount ephemeral to /var/lib/docker
      Requires=format-ephemeral.service
      After=format-ephemeral.service
      Before=docker.service
      [Mount]
      What=/dev/xvdf
      Where=/var/lib/docker
      Type=btrfs
```

Following are some details on the preceding `cloud-config` unit file:

- The Format-ephemeral service takes care of formatting the filesystem as `btrfs`
- The Mount service takes care of mounting the new volume in `/var/lib/docker` before `docker.service` is started

We can start the CoreOS node with the preceding `cloud-config` with extra EBS volume using the following commands:

```
aws ec2 run-instances --image-id ami-85ada4b5 --count 1 --instance-
type t2.micro --key-name "smakam-oregon" --security-groups "coreos-test"
--user-data file://cloud-config-mntdocker.yaml --block-device-mappings
"[{\"DeviceName\":\"/dev/sdf\",\"Ebs\":{\"DeleteOnTermination\":false,\"V
olumeSize\":8,\"VolumeType\":\"gp2\"}}]"
```

The preceding command creates a single-node CoreOS cluster with one extra volume of 8 GB. The new volume is mounted as `/var/lib/docker` with the `btrfs` filesystem. The `/dev/sdf` directory gets mounted in the CoreOS system as `/dev/xvdf`, so the mount file uses `/dev/xvdf`.

The following is the partition table in the node with the preceding `cloud-config`:

```
core@ip-172-31-46-214 ~ $ df -k
Filesystem     1K-blocks    Used Available Use% Mounted on
devtmpfs          497168       0    497168   0% /dev
tmpfs             510100       0    510100   0% /dev/shm
tmpfs             510100     324    509776   1% /run
tmpfs             510100       0    510100   0% /sys/fs/cgroup
/dev/xvda9       5706380   20816   5409236   1% /
/dev/xvda3       1007760  354076    601668  38% /usr
tmpfs             510100       0    510100   0% /tmp
tmpfs             510100       0    510100   0% /media
/dev/xvda6        110576      60    101344   1% /usr/share/oem
/dev/xvdf        8388608  180316   7382820   3% /var/lib/docker
```

As we can see, there is a new 8 GB partition where `/var/lib/docker` is mounted.

The following output shows you that the docker filesystem is using the `btrfs` storage driver as we requested:

```
core@ip-172-31-46-214 ~ $ docker info
Containers: 1
Images: 12
Storage Driver: btrfs
 Build Version: Btrfs v3.17.1
```

Mounting NFS storage

We can mount a volume on a CoreOS node using NFS. NFS allows a shared storage mechanism where all CoreOS nodes in the cluster can see the same data. This approach can be used for Container data persistence when Containers are moved across nodes. In the following example, we run the NFS server in a Linux server and mount this volume in a CoreOS node running in the Vagrant environment.

The following are the steps to set up NFS mounting on the CoreOS node:

1. Start the NFS server and export directories that are to be shared.
2. Set up the CoreOS `cloud-config` to start `rpc-statd.service`. Mount services also need to be started in the `cloud-config` to mount the necessary NFS directories to local directories.

Setting up NFS server

Start the NFS server. I had set up my Ubuntu 14.04 machine as an NFS server. The following are the steps that I performed to set up the NFS server:

1. Install the NFS server:

   ```
   sudo apt-get install nfs-kernel-server
   ```

2. Create an NFS directory with the appropriate owner:

   ```
   sudo mkdir /var/nfs
   sudo chown core /var/nfs (I have created a core user)
   ```

3. Export the `NFS` directory to the necessary nodes. In my case, `172.17.8.[101-103]` are the IP addresses of the CoreOS cluster. Create `/etc/exports` with the following commands:

   ```
   /var/nfs    172.17.8.101(rw,sync,no_root_squash,no_subtree_check)
   /var/nfs    172.17.8.102(rw,sync,no_root_squash,no_subtree_check)
   /var/nfs    172.17.8.103(rw,sync,no_root_squash,no_subtree_check)
   ```

4. Start the NFS server:

   ```
   sudo exportfs -a
   sudo service nfs-kernel-server start
   ```

Note: NFS is pretty sensitive to **UserID** (**UID**) and **Group ID** (**GID**) checks, and write access from the client machine won't work unless this is properly set up. It is necessary for the UID and GID of the client user to match with the UID and GID of the directory setup in the server. Another option is to set the `no_root_squash` option (as in the preceding example) so that the root user from the client can make modifications as the UserID in the server.

As shown in the following command, we can see the directory exported after making the necessary configuration:

```
smakam14@jungle1:~$ showmount -e 172.17.8.110
Export list for 172.17.8.110:
/var/nfs 172.17.8.111,172.17.8.103,172.17.8.102,172.17.8.101
```

Setting up the CoreOS node as a client for the NFS

The following `cloud-config` can be used to mount remote `/var/nfs in /mnt/data` in all the nodes of the CoreOS cluster:

```
#cloud-config

write-files:
  - path: /etc/conf.d/nfs
    permissions: '0644'
    content: |
      OPTS_RPC_MOUNTD=""

coreos:
  etcd2:
    discovery: <yourtoken>
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  fleet:
    public-ip: $public_ipv4
  flannel:
    interface: $public_ipv4
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
    - name: rpc-statd.service
      command: start
      enable: true
    - name: mnt-data.mount
      command: start
      content: |
        [Mount]
        What=172.17.8.110:/var/nfs
        Where=/mnt/data
        Type=nfs
        Options=vers=3,sec=sys,noauto
```

In the preceding config, `cloud-config`, `rpc-statd.service` is necessary for the NFS client service and `mnt-data.mount` is necessary to mount the NFS volume in the `/mnt/data` local directory.

The following output is in one of the CoreOS nodes that have done the NFS mount. As we can see, the NFS mount is successful:

```
core@core-01 ~ $ mount -t nfs
172.17.8.110:/var/nfs on /mnt/data type nfs (rw,relatime,vers=3,rsize=1048576,wsize=1048576,namlen=255,
hard,proto=tcp,timeo=600,retrans=2,sec=sys,mountaddr=172.17.8.110,mountvers=3,mountport=41728,mountprot
o=udp,local_lock=none,addr=172.17.8.110)
```

After this step, any CoreOS nodes in the cluster can read and write from `/mnt/data`.

The container filesystem

Containers use the CoW filesystem to store Container images. The following are some characteristics of the CoW filesystem:

- Multiple users/processes can share the same data as if they have their own copy of the data.
- If data is changed by any one process or user, a new copy of the data is made for this process/user only at that point.
- Multiple running containers share the same set of files till changes are made to the files. This makes starting the containers really fast.

These characteristics allow the Container filesystem operations to be really fast. Docker supports multiple storage drivers that are capable of CoW. Each OS chooses a default storage driver. Docker provides you with an option to change the storage driver. To change the storage driver, we need to specify the storage driver in `/etc/default/docker` and restart the Docker daemon:

```
DOCKER_OPTS="--storage-driver=<driver>"
```

The major supported storage drivers are `aufs`, `devicemapper`, `btrfs`, and `overlay`. We need to make sure that the storage driver is supported by the OS on which Docker is installed before changing the Storage driver.

Storage drivers

The storage driver is responsible for managing the filesystem. The following table captures the differences between major storage drivers supported by Docker:

Property	**AUFS**	**Device mapper**	**BTRTS**	**OverlayFS**	**ZFS**
File/block	File-based	Block-based	File-based	File-based	File-based
Linux kernel support	Not in the main kernel	Present in the main kernel	Present in the main kernel	Present in the main kernel > 3.18	Not in the main kernel
OS	Ubuntu default	Red Hat		Red Hat	Solaris
Performance	Not suitable to write big files; useful for PaaS scenarios	First write slow	Updating a lot of small files can cause low performance	Better than AUFS	Takes up a lot of memory

A storage driver needs to be chosen based on the type of workload, the need for availability in the main Linux kernel, and the comfort level with a particular storage driver.

The following output shows the default AUFS storage driver used by Docker running on the Ubuntu system:

```
root@jungle1:~# docker info
Containers: 3
Images: 167
Storage Driver: aufs
 Root Dir: /var/lib/docker/aufs
 Backing Filesystem: extfs
 Dirs: 173
 Dirperm1 Supported: true
```

The following output shows Docker using the Overlay driver in the CoreOS node. CoreOS was using `btrfs` sometime back. Due to `btrfs` stability issues, they moved to the Overlay driver recently.

```
core@core-01 ~ $ docker info
Containers: 0
Images: 0
Storage Driver: overlay
 Backing Filesystem: extfs
```

The `/var/lib/docker` directory is where the container metadata and volume data is stored. The following important information is stored here:

- Containers: The container metadata
- Volumes: The host volumes
- Storage drivers such as aufs and device mapper: These will contain diffs and layers

The following screenshot shows the directory output in the Ubuntu system running Docker:

```
root@sreeubuntu14-VirtualBox1:/var/lib/docker/0.0# ls
aufs  containers  graph  init  linkgraph.db  network  repositories-aufs  tmp  trust  volumes
```

Docker and the Union filesystem

Docker images make use of the Union filesystem to create an image composed of multiple layers. The Union filesystem makes use of the CoW techniques. Each layer is like a snapshot of the image with a particular change. The following example shows you the image layers of an Ubuntu docker image:

```
smakam14@jungle1:~/docker/apache$ docker history ubuntu:14.04
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
d2a0ecffe6fa        3 months ago        /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
29460ac93442        3 months ago        /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/   1.895 kB
b670fb0c7ecd        3 months ago        /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic   194.5 kB
33e4dde6b9cf        3 months ago        /bin/sh -c #(nop) ADD file:c8f078961a543cdefa   188.2 MB
```

Each layer shows the operations done on the base layer to get this new layer.

To illustrate the layering, let's take this base Ubuntu image and create a new container image using the following Dockerfile:

```
FROM ubuntu:14.04
MAINTAINER Sreenivas Makam <smxxxx@yahoo.com>

# Install apache2
RUN apt-get install -y apache2

EXPOSE 80
ENTRYPOINT ["/usr/sbin/apache2ctl"]
CMD ["-D", "FOREGROUND"]
```

Build a new Docker image:

```
docker build -t="smakam/apachetest"
```

Let's look at the layers of this new screenshot:

```
smakam14@jungle1:~/docker/apache$ docker history smakam/apachetest
IMAGE               CREATED             CREATED BY                                      SIZE                COMMENT
6e072b7a6bb5        About an hour ago   /bin/sh -c #(nop) CMD ["-D" "FOREGROUND"]       0 B
ffc71f2a556c        About an hour ago   /bin/sh -c #(nop) ENTRYPOINT &{["/usr/sbin/ap   0 B
317aef41c6e0        About an hour ago   /bin/sh -c #(nop) EXPOSE 80/tcp                 0 B
cf8db655e5f3        About an hour ago   /bin/sh -c apt-get install -y apache2           14.41 MB
406fdcdfe5ee        About an hour ago   /bin/sh -c #(nop) MAINTAINER Sreenivas Makam    0 B
d2a0ecffe6fa        3 months ago        /bin/sh -c #(nop) CMD ["/bin/bash"]             0 B
29460ac93442        3 months ago        /bin/sh -c sed -i 's/^#\s*\(deb.*universe\)$/   1.895 kB
b670fb0c7ecd        3 months ago        /bin/sh -c echo '#!/bin/sh' > /usr/sbin/polic   194.5 kB
83e4dde6b9cf        3 months ago        /bin/sh -c #(nop) ADD file:c8f078961a543cdefa   188.2 MB
```

The first four layers are the ones created from the Dockerfile, and the last four layers are part of the Ubuntu 14.04 base image. In case you have the Ubuntu 14.04 image in your system and try to download `smakam/apachetest`, only the first four layers would be downloaded as the other layers will already be present in the host machine and can be reused. This layer reuse mechanism allows a faster download of Docker images from the Docker hub as well as efficient storage of Docker images in the Container filesystem.

Container data

Container data is not part of the Container filesystem and is stored in the host filesystem where Container runs. Container data can be used to store data that needs to be manipulated frequently, such as a database. Container data typically needs to be shared between multiple Containers.

Docker volumes

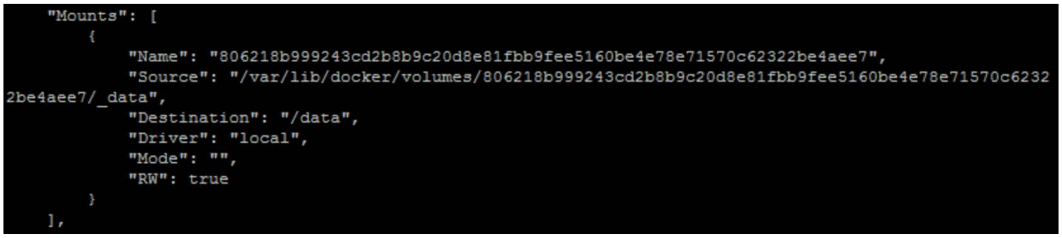
Changes made in the container are stored as part of the Union filesystem. If we want to save some data outside the scope of the container, volumes can be used. Volumes are stored as part of the host filesystem and it gets mounted in the Container. When container changes are committed, volumes are not committed as they reside outside the Container filesystem. Volumes can be used to share the source code with the host filesystem, maintain persistent data like a database, share data between containers, and function as a scratch pad for the container. Volumes give better performance over the Union filesystem for applications such as databases where we need to do frequent read and write operations. Using volumes does not guarantee Container data persistence. Using data-only Containers is an approach to maintain the persistence and share data across Containers. There are other approaches such as using shared and distributed storage to persist Container data across hosts.

Container volume

The following example starts the Redis container with the `/data` volume:

```
docker run -d --name redis -v /data redis
```

If we run Docker to inspect Redis, we can get details about the volumes mounted by this container, as can be seen in the following screenshot:



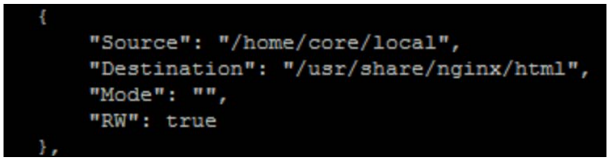
The `Source` directory is the directory in the host machine and `Destination` is the directory in the Container.

Volumes with the host mount directory

The following is an example of code sharing with the mounting host directory using Volume:

```
docker run -d --name nginxpersist -v /home/core/local:/usr/share/nginx/
html -p ${COREOS_PUBLIC_IPV4}:8080:80 nginx
```

If we perform `docker inspect nginxpersist`, we can see both the host directory and the container mount directory:



In the host machine, code development can be done in the `/home/core/local` location, and any code change in the host machine automatically reflects in the container.

As the host directory can vary across hosts, this makes Containers unportable and Dockerfile does not support the host mount option.

A data-only container

Docker has support for a data-only container that is pretty powerful. Multiple containers can inherit the volume from a data-only container. The advantage with a data-only container over regular host-based volume mounting is that we don't have to worry about host file permissions. Another advantage is a data-only container can be moved across hosts, and some of the recent Docker volume plugins take care of moving the volume data when the container moves across hosts.

The following example shows you how volumes can be persisted when containers die and restart.

Let's create a volume container, `redisvolume`, for `redis` and use this volume in the `redis1` container. The `hellocounter` container counts the number of web hits and uses the `redis` container for counter-persistence:

```
docker run -d --name redisvolume -v /data redis
docker run -d --name redis1 --volumes-from redisvolume redis
docker run -d --name hello1 --link redis1:redis -p 5000:5000 smakam/
hellocounter python app.py
```

Let's see the running containers:

```
smakam14@jungle1:~$ docker ps -a
CONTAINER ID        IMAGE                  COMMAND                CREATED             STATUS
     PORTS                    NAMES
8619b2203692        smakam/hellocounter    "python app.py"        2 minutes ago       Up 2 minutes
     0.0.0.0:5000->5000/tcp   hello1
2f0957d505a7        redis                  "/entrypoint.sh redi   2 minutes ago       Up 2 minutes
     6379/tcp                 redis1
313f0ea142d8        redis                  "/entrypoint.sh redi   2 minutes ago       Up 2 minutes
     6379/tcp                 redisvolume
```

Let's access the hellocounter container multiple times using curl, as shown in the following image:

```
smakam14@jungle1:~$ curl localhost:5000
Hello World! I have been seen 5 times.s
```

Now, let's stop this container and restart another container using the following commands. The new redis container, `redis2`, still uses the same `redisvolume` container:

```
docker stop redis1 hello1
docker rm redis1 hello1
docker run -d --name redis2 --volumes-from redisvolume redis
docker run -d --name hello2 --link redis2:redis -p 5001:5000 smakam/
hellocounter python app.py
```

If we try to access the hellocounter container using port `5001`, we will see that the counter starts from `6` as the previous value `5` is persisted in the database even though we have stopped that container and restarted a new redis container:

```
smakam14@jungle1:~$ curl localhost:5001
Hello World! I have been seen 6 times.s
```

A data-only container can also be used to share data between containers. An example use case could be a web container writing a log file and a log processing container processing the log file and exporting it to a central server. Both the web and log containers can mount the same volume with one container writing to the volume and another reading from the volume.

To back up the `redisvolume` container data that we created, we can use the following command:

```
docker run --volumes-from redisvolume -v $(pwd):/backup ubuntu tar cvf /
backup/backup.tar /data
```

This will take `/data` from `redisvolume` and back up the content to `backup.tar` in the current host directory using an Ubuntu container to do the backup.

Removing volumes

As part of removing a container, if we use the `docker rm -v` option, the volume will be automatically deleted. If we forget to use the `-v` option, volumes will be left dangling. This has the disadvantage that the space allocated in the host machine for the volume will be unused and not removed.

Docker until release 1.7 does not yet have a native solution to handle dangling volumes. There are some experimental containers available to clean up dangling volumes. I use this test Container, `martin/docker-cleanup-volumes`, to clean up my dangling volumes. First, we can determine the dangling volumes using the `dry-run` option. The following is an example that shows four dangling volumes and one volume that is in use:

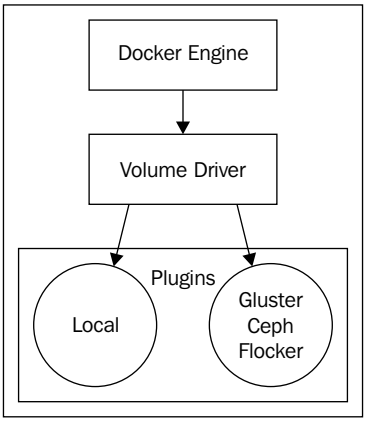
```
core@core-01 ~ $ docker run -v /var/run/docker.sock:/var/run/docker.sock -v /var/lib/docker:/var/lib/doc
ker --rm martin/docker-cleanup-volumes --dry-run

Delete unused volume directories from /var/lib/docker/volumes
Would have deleted fe982bc8078377f2d5393e070c77f741f7d57b476476ceb3f8c59664a84f62ae
Would have deleted 43e2b12d46859fd05c79c21fdcce63e20520a8863b6092be9166181bfc97d634
Would have deleted 20acbaf2b3ebd7f6acd8d555f9848a19c6e93a2a8dec620876e47fae496e7623
Would have deleted de7d0a7144f6b027a1cd540302cc1b32f3780c60633804c41d583c88cf706601
In use 1f2ad955f8a14334af7bd508b494eb6d51275458192e07bab0c2376f21afd9ec
```

If we remove the `dry-run` option, dangling volumes will be deleted, as shown in the following image:

```
core@core-01 ~ $ docker run -v /var/run/docker.sock:/var/run/docker.sock -v /var/lib/docker:/var/lib/doc
ker --rm martin/docker-cleanup-volumes

Delete unused volume directories from /var/lib/docker/volumes
Deleting fe982bc8078377f2d5393e070c77f741f7d57b476476ceb3f8c59664a84f62ae
Deleting 43e2b12d46859fd05c79c21fdcce63e20520a8863b6092be9166181bfc97d634
Deleting 20acbaf2b3ebd7f6acd8d555f9848a19c6e93a2a8dec620876e47fae496e7623
Deleting de7d0a7144f6b027a1cd540302cc1b32f3780c60633804c41d583c88cf706601
In use 1f2ad955f8a14334af7bd508b494eb6d51275458192e07bab0c2376f21afd9ec
```

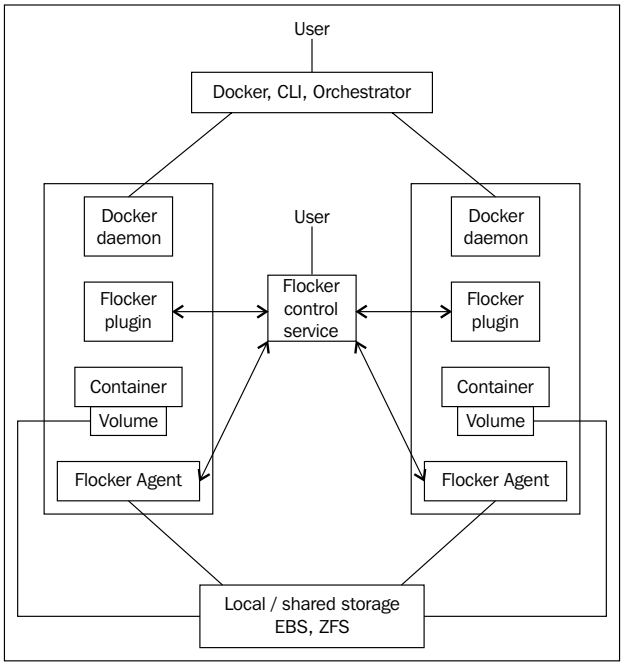
The Docker Volume plugin

Like the Network plugin for Docker, the Volume plugin extends storage functionality for Docker containers. Volume plugins provide advanced storage functionality such as volume persistence across nodes. The following figure shows you the volume plugin architecture where the Volume driver exposes a standard set of APIs, which plugins can implement. GlusterFS, Flocker, Ceph, and a few other companies provide Docker volume plugins. Unlike the Docker networking plugin, Docker does not have a native volume plugin and relies on plugins from external vendors:



Flocker

Docker data volumes are tied to a single node where the Container is created. When Containers are moved across nodes, data volumes don't get moved. Flocker addresses this issue of moving the data volumes along with the Container. The following figure shows you all the important blocks in the Flocker architecture:



The following are some internals of the Flocker implementation:

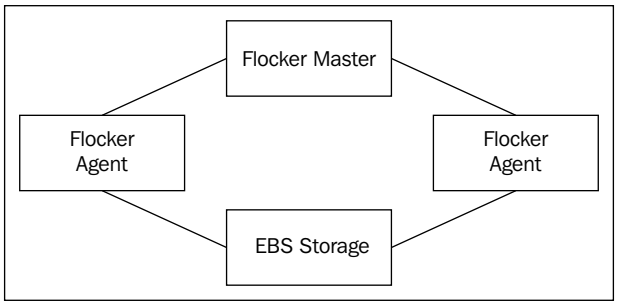
- The Flocker agent runs in each node and takes care of talking to the Docker daemon and the Flocker control service.
- The Flocker control service takes care of managing the volumes as well as the Flocker cluster.
- Currently supported backend storage includes Amazon AWS EBS, Rackspace block storage, and EMC ScaleIO. Local storage using ZFS is available on an experimental basis.
- Both the REST API and Flocker CLI are used to manage volumes as well as Docker containers.
- Docker can manage volumes using Flocker as a data volume plugin.

- The Flocker plugin will take care of managing data volumes, which includes migrating the volume associated with the Container when the Container moves across hosts.
- Flocker will use the Container networking technology to talk across hosts—this can be native Docker networking or Docker networking plugins such as Weave.

In the next three examples, we will illustrate how Flocker achieves Container data persistence in different environments.

Flocker volume migration using AWS EBS as a backend

This example will illustrate data persistence using AWS EBS as a storage backend. In this example, we will create three Linux nodes in the AWS cloud. One node will serve as the Flocker master running the control service and the other two nodes will run Flocker agents running the containers and mounting the EBS storage. Using these nodes, we will create a stateful Container and demonstrate Container data persistence on Container migration. The example will use a hellocounter container with the redis container backend and illustrates data persistence when the redis counter is moved across hosts. The following figure shows you how the master and agents are tied to the EBS backend:



I followed the procedure mentioned on the Flocker web page—`https://docs.clusterhq.com/en/1.4.0/labs/installer.html`—for this example.

The following are the summary of steps to setup Flocker volume migration using AWS EBS:

1. It's necessary to have an AWS account to create VMs running Docker containers and Flocker services.
2. For the execution of frontend Flocker commands, we need a Linux host. In my case, it's a Ubuntu 14.04 VM.
3. Install Flocker frontend tools on the Linux host using Flocker scripts.

4. Install the Flocker control service on the control node and Flocker agents on the slave nodes using Flocker scripts.
5. At this point, we can create containers on slave nodes with a data volume and migrate containers keeping the volume persistent.

The following are the relevant outputs after installing the Flocker frontend tools and Flocker control service and agents.

This is the version of the Flocker frontend tools:

```
smakam14@jungle1:~/aws1$ uft-flocker-ca --version
1.4.0
```

The Flocker node list shows the two AWS nodes that will run Flocker agents:

```
smakam14@jungle1:~/clusters/test$ uft-flocker-volumes list-nodes
SERVER     ADDRESS
69d25ff3   10.0.206.63
a6a4b5b9   10.0.115.120
```

The following output shows you the Flocker volume list. Initially, there are no volumes:

```
smakam14@jungle1:~/clusters/test$ uft-flocker-volumes list
DATASET    SIZE    METADATA    STATUS    SERVER
```

Let's look at the main processes in the master node. We can see the control service running in the following screenshot:

```
root@ip-10-0-132-11:~# ps -eaf|grep flocker
root     29703     1  1 16:51 ?        00:00:04 /opt/flocker/bin/python /usr/sbin/flocker-control -p
 tcp:4523 -a tcp:4524 --logfile=/var/log/flocker/flocker-control.log
```

Let's look at the main processes in the slave node. Here, we can see the Flocker agents and the Flocker docker plugin running:

```
root@ip-10-0-206-63:~# ps -eaf|grep flocker
root     29764     1  1 16:51 ?        00:00:08 /opt/flocker/bin/python /usr/sbin/flocker-dataset-ag
ent --logfile=/var/log/flocker/flocker-dataset-agent.log
root     29782     1  1 16:51 ?        00:00:07 /opt/flocker/bin/python /usr/sbin/flocker-container-
agent --logfile=/var/log/flocker/flocker-container-agent.log
root     30374     1  0 16:52 ?        00:00:04 /usr/bin/python /usr/bin/twistd -noy /opt/flocker/li
b/python2.7/site-packages/flockerdockerplugin/flockerdockerplugin.tac
```

Let's create a hellocounter container with the redis container backend on a particular slave node, update the counter in the database, and then move the container to demonstrate that the data volume gets persisted as the container is moved.

Let's first set up some shortcuts:

```
NODE1="52.10.201.177" (this public ip address corresponds to the private
address shown in flocker list-nodes output)
NODE2="52.25.14.152"
KEY="keylocation "
```

Let's start the `hellocontainer` and `redis` containers on `node1`:

```
ssh -i $KEY root@$NODE1 docker run -d -v demo:/data --volume-
driver=flocker --name=redis redis:latest
ssh -i $KEY root@$NODE1 docker run -d -e USE_REDIS_HOST=redis --link
redis:redis -p 80:5000 --name=hellocounter smakam/hellocounter
```

Let's look at the volumes created and attached at this point. 100 GB EBS volume is attached to slave `node1` at this point:

```
smakam14@jungle1:~/clusters/test$ uft-flocker-volumes list
DATASET                              SIZE      METADATA    STATUS         SERVER
31a3f409-b4c4-491d-b6ce-15b998881346   100.00G   name=demo   attached ✅   69d25ff3 (10.0.206.63)
```

From the following output, we can see the two containers running in `node1`:

```
root@ip-10-0-206-63:~# docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED              STATUS
        PORTS                    NAMES
3577d661eff8        smakam/hellocounter   "/bin/sh -c 'python a"   About a minute ago   Up About a m
inute   0.0.0.0:80->5000/tcp   hellocounter
38b86d2c6de0        redis:latest          "/entrypoint.sh redis"   5 minutes ago        Up 4 minutes
        6379/tcp                 redis
```

Let's create some entries in the database now. The counter value is currently at `6`, as shown in the following screenshot:

```
smakam14@jungle1:~/aws1$ curl 54.149.227.108
Hello World! I have been seen 6 times.smakam1
```

Now, let's remove the containers in `NODE1` and create the `hellocounter` and `redis` containers in `NODE2`:

```
ssh -i $KEY root@$NODE1 docker stop hellocounter
ssh -i $KEY root@$NODE1 docker stop redis
```

```
ssh -i $KEY root@$NODE1 docker rm -f hellocounter
ssh -i $KEY root@$NODE1 docker rm -f redis
ssh -i $KEY root@$NODE2 docker run -d -v demo:/data --volume-
driver=flocker --name=redis redis:latest
ssh -i $KEY root@$NODE2 docker run -d -e USE_REDIS_HOST=redis --link
redis:redis -p 80:5000 --name=hellocounter smakam/hellocounter
```

As we can see, the volume has migrated to the second slave node:

```
smakam14@jungle1:~/clusters/test$ uft-flocker-volumes list
DATASET                              SIZE      METADATA    STATUS         SERVER
31a3f409-b4c4-491d-b6ce-15b998881346   100.00G   name=demo   attached ✓   a6a4b5b9 (10.0.115.120)
```

Let's look at the containers in `node2`:

```
root@ip-10-0-115-120:~# docker ps
CONTAINER ID        IMAGE                 COMMAND                  CREATED              STATUS
        PORTS                    NAMES
12b311b26b34        smakam/hellocounter   "/bin/sh -c 'python a"   13 seconds ago       Up 11 second
s       0.0.0.0:80->5000/tcp     hellocounter
1a72c9aa41ce        redis:latest          "/entrypoint.sh redis"   About a minute ago   Up About a m
inute   6379/tcp                 redis
```

Now, let's check whether the data is persistent:

```
smakam14@jungle1:~/aws1$ curl 52.10.64.125
Hello World! I have been seen 7 times.smak
```

As we can see from the preceding output, the counter value starts from the previous count of `6` and is incremented to `7`, which shows that the redis database is persistent when the redis container is moved across the nodes.

Flocker volume migration using the ZFS backend

This example will illustrate data persistence using ZFS as a storage backend and Vagrant Ubuntu cluster. ZFS is an open source filesystem that focuses on data integrity, replication, and performance. I followed the procedure at `https://docs.clusterhq.com/en/1.4.0/using/tutorial/vagrant-setup.html` to set up a two-node Vagrant Ubuntu Flocker cluster and at `https://docs.clusterhq.com/en/1.4.0/using/tutorial/volumes.html` to try out the sample application that allows container migration with the associated volume migration. The sample application uses the MongoDB container for data storage and illustrates data persistence.

The following are the summary of steps to setup Flocker volume migration using ZFS backend:

1. Install Flocker client tools and the `mongodb` client in the client machine. In my case, this is a Ubuntu 14.04 VM.
2. Create a two-node Vagrant Ubuntu cluster. As part of the cluster setup, Flocker services are started in each of the nodes and this includes control and agent services.
3. Start the flocker-deploy script starting the `mongodb` container on `node1`.
4. Start the `mongodb` client and write some entries in `node1`.
5. Start the flocker-deploy script moving the `mongodb` container from `node1` to `node2`.
6. Start the `mongbdb` client to `node2` and check whether the data is retained.

After starting the two-node Vagrant cluster, let's check the relevant Flocker services.

`Node1` has both the Flocker control and agent services running, as shown in the following screenshot:

```
[vagrant@node1 ~]$ ps -eaf|grep flocker
root       763     1  3 14:57 ?        00:00:34 /opt/flocker/bin/python /usr/sbin/flocker-control -p systemd:INET:0 -a system
d:INET:1 --journald
root       766     1  5 14:57 ?        00:00:51 /opt/flocker/bin/python /usr/sbin/flocker-dataset-agent --journald
root      2176     1  9 14:59 ?        00:01:17 /opt/flocker/bin/python /usr/sbin/flocker-container-agent --journald
```

`Node2` has only the Flocker agent service running and is being managed by `Node1`, as shown in the following screenshot:

```
[vagrant@node2 ~]$ ps -eaf|grep flocker
root       768     1  5 15:08 ?        00:01:47 /opt/flocker/bin/python /usr/sbin/flocker-dataset-agent --journald
root      2194     1  7 15:10 ?        00:02:24 /opt/flocker/bin/python /usr/sbin/flocker-container-agent --journald
```

Let's look at the Flocker node list; this shows the two nodes:

```
sreeni@ubuntu:~/flocker-tutorial$ flocker-volumes --control-service=172.16.255.250 list-nodes
SERVER     ADDRESS
430e9391   172.16.255.251
43a06d55   172.16.255.250
```

Let's deploy the `mongodb` container on `node1` as follows:

```
sreeni@ubuntu:~/flocker-tutorial$ flocker-deploy 172.16.255.250 volume-deployment.yml volume-appli
cation.yml
```

Let's look at the volume list. As we can see, the volume is attached to `node1`:

```
sreeni@ubuntu:~/flocker-tutorial$ flocker-volumes  --control-service=172.16.255.250 list
DATASET                              SIZE         METADATA                      STATUS         S
ERVER
e83b21be-2d1a-4bd5-bee3-43dfa025f6e8   <no quota>   name=mongodb-volume-example   attached ✓   43a
06d55 (172.16.255.250)
```

The following output shows you the container in `node1`:

```
[vagrant@node1 ~]$ docker ps
CONTAINER ID        IMAGE                     COMMAND                CREATED             STATUS              PORTS
          NAMES
9691d892a67c        clusterhq/mongodb:latest   "/bin/sh -c '/home/mo"   4 minutes ago       Up 4 minutes        0.0.0.0:27017
->27017/tcp   flocker--mongodb-volume-example
```

Let's add some data to `mongodb`:

```
> use example;
switched to db example
> db.records.insert({"the data": "it moves"})
```

Now, let's redeploy the container to `node2`:

```
sreeni@ubuntu:~/flocker-tutorial$ flocker-deploy 172.16.255.250 volume-deployment-moved.yml volume
-application.yml
```

Let's look at the volume output. As we can see, the volume is moved to `node2`:

```
sreeni@ubuntu:~/flocker-tutorial$ flocker-volumes --control-service=172.16.255.250 list
DATASET                              SIZE         METADATA                      STATUS         SERVER
e83b21be-2d1a-4bd5-bee3-43dfa025f6e8   <no quota>   name=mongodb-volume-example   attached ✓   430e9391 (172.16.255.251)
```

As we can see from the following output, the `mongodb` content, `the data`, is preserved:

```
> db.records.find({})
{ "_id" : ObjectId("5621002d77f1111f74cbc2d8"), "the data" : "it moves" }
```

Flocker on CoreOS with an AWS EBS backend

Flocker has recently integrated with CoreOS on an experimental basis with the AWS EBS backend storage. I followed the procedures at https://github.com/ clusterhq/ flocker-coreos and `https://clusterhq.com/2015/09/01/flocker-runs-on-coreos/` for this example. I had some issues with getting version 1.4.0 of the Flocker tools to work with CoreOS nodes. The 1.3.0 version of tools (`https://docs.clusterhq.com/en/1.3.0/labs/installer.html`) worked fine.

In this example, we have illustrated Container data persistence on the CoreOS cluster with Docker using the Flocker plugin.

The following are the summary of steps to setup Flocker volume migration on CoreOS cluster running on AWS:

1. Create a CoreOS cluster using AWS Cloudformation with the template specified by Flocker along with a newly created discovery token.
2. Create `cluster.yml` with the node IP and access details.
3. Start the Flocker script to configure the CoreOS nodes with the Flocker control service as well as Flocker agents. Flocker scripts also take care of replacing the default Docker binary in the CoreOS node with the Docker binary that supports the volume plugin.
4. Check that Container migration is working fine with data persistence.

I used the following Cloudformation script to create a CoreOS cluster using the template file from Flocker:

```
aws cloudformation create-stack     --stack-name coreos-
test1     --template-body file://coreos-stable-flocker-hvm.template
--capabilities CAPABILITY_IAM     --tags Key=Name,Value=CoreOS
--parameters      ParameterKey=DiscoveryURL,ParameterValue="your token"
ParameterKey=KeyPair,ParameterValue="your keypair"
```

The following are the details of the CoreOS cluster that has three nodes:

```
ip-172-31-35-201 ~ # fleetctl list-machines
MACHINE         IP              METADATA
45a58620...     172.31.35.199   -
7a027bb8...     172.31.35.201   -
aa3244a1...     172.31.35.200   -
```

The following are the old and new Docker versions installed. Docker version 1.8.3 supports the Volume plugin:

```
ip-172-31-35-201 ~ # docker --version
Docker version 1.6.2, build 7c8fca2-dirty
ip-172-31-35-201 ~ # /root/bin/docker --version
Docker version 1.8.3, build f4bf5c7
```

The following is the CoreOS version in the node:

```
ip-172-31-35-201 ~ # cat /etc/os-release
NAME=CoreOS
ID=coreos
VERSION=723.3.0
```

The following output shows the Flocker node list with three CoreOS nodes running Flocker:

```
sreeni@ubuntu:~/clusters/test$ flocker-volumes list-nodes
SERVER     ADDRESS
74a07fe1   172.31.35.200
a1e31ce0   172.31.35.199
4952522a   172.31.35.201
```

I tried the same `hellocounter` example as mentioned in the previous section, and the volume moved automatically across the nodes. The following output shows the volume initially attached to `node1` and later moved to `node2` as part of the Container move.

This is the volume attached to `node1`:

```
sreeni@ubuntu:~/clusters/test$ flocker-volumes list
DATASET                              SIZE      METADATA    STATUS         SERVER
894c178f-bcc6-4afc-9ac3-9b027d22feb5   100.00G   name=demo   attached ✓   4952522a (172.31.35.201)
```

This is the volume attached to `node2`:

```
sreeni@ubuntu:~/clusters/test$ flocker-volumes list
DATASET                              SIZE      METADATA    STATUS         SERVER
894c178f-bcc6-4afc-9ac3-9b027d22feb5   100.00G   name=demo   attached ✓   74a07fe1 (172.31.35.200)
```

According to the Flocker documentation, they have a plan to support the ZFS backend on CoreOS at some point, to allow us to use local storage instead of AWS EBS. It's still not certain if CoreOS will support ZFS natively.

Flocker recent additions

Flocker added the following functionality recently, as of November 2015:

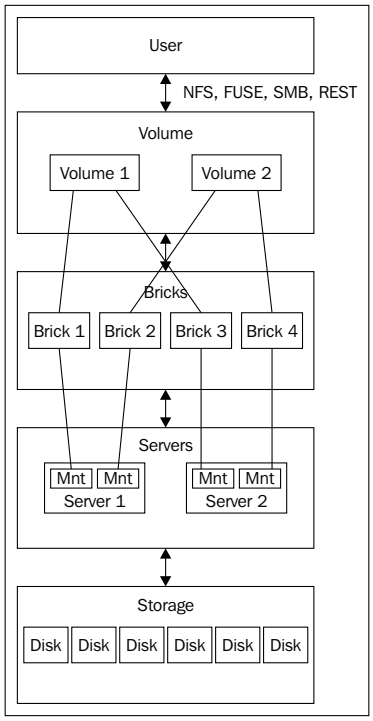
- The Flocker volume hub (`https://clusterhq.com/volumehub/`) manages all Flocker volumes from a central location.
- Flocker dvol (`https://clusterhq.com/dvol/`) provides you with a Git-like functionality for data volumes. This can help manage databases such as a codebase.

GlusterFS

GlusterFS is a distributed filesystem where the storage is distributed across multiple nodes and presented as a single unit. GlusterFS is an open source project and works on any kind of storage hardware. Red Hat has acquired Gluster, which started GlusterFS. The following are some properties of GlusterFS:

- Multiple servers with their associated storage are joined to a GlusterFS cluster using the peering relationship.
- GlusterFS can work on top of the commodity storage as well as SAN.
- By avoiding a central metadata server and using a distributed hashing algorithm, GlusterFS clusters are scalable and can expand into very large clusters.
- Bricks are the smallest component of storage from the GlusterFS perspective. A brick consists of mount points created from a storage disk with a base filesystem. Bricks are tied to a single server. A single server can have multiple bricks.
- Volumes are composed of multiple bricks. Volumes are mounted to the client device as a mount directory.
- Major volume types are distributed, replicated, and striped. A distributed volume type allows the distributing of files across multiple bricks. A replicated volume type allows multiple replicas of the file, which is useful from a redundancy perspective. A striped volume type allows the splitting of a large file into multiple smaller files and distributing them across the bricks.
- GlusterFS supports multiple access methods to access the GlusterFS volume, and this includes native FUSE-based access, SMB, NFS, and REST.

The following figure shows you the different layers of GlusterFS:



Setting up a GlusterFS cluster

In the following example, I have set up a two-node GlusterFS 3.5 cluster with each server running a Ubuntu 14.04 VM. I have used the GlusterFS server node as the GlusterFS client as well.

The following is a summary of steps to setup a GlusterFS cluster:

1. Install the GlusterFS server on both the nodes and the client software on one of the nodes in the cluster.
2. GlusterFS nodes must be able to talk to each other. We can either set up DNS or use a static `/etc/hosts` approach for the nodes to talk to each other.
3. Turn off firewalls, if needed, for the servers to be able to talk to each other.
4. Set up GlusterFS server peering.
5. Create bricks.

6. Create volumes on top of the created bricks.
7. In the client machine, mount the volumes to mountpoint and start using GlusterFS.

The following commands need to be executed in each server. This will install the GlusterFS server component. This needs to be executed on both the nodes:

```
sudo apt-get install software-properties-common
sudo add-apt-repository ppa:gluster/glusterfs-3.5
sudo apt-get update
sudo apt-get install glusterfs-server
```

The following command will install the GlusterFS client. This is necessary only in `node1`:

```
sudo apt-get install glusterfs-client
```

Set up `/etc/hosts` to allow nodes to talk to each other:

```
192.168.56.102  gluster1
192.168.56.101  gluster2
```

Disable the firewall:

```
sudo iptables -I INPUT -p all -s 192.168.56.102 -j ACCEPT
sudo iptables -I INPUT -p all -s 192.168.56.101 -j ACCEPT
```

Create the replicated volume and start it:

```
sudo gluster volume create volume1 replica 2 transport tcp gluster1:/
gluster-storage gluster2:/gluster-storage force (/gluster-storage is the
brick in each node)
sudo gluster volume start volume1
```

Set up a server probe in each node. The following command is for `Node1`:

```
sudo gluster peer probe gluster2
```

The following command is for `Node2`:

```
sudo gluster peer probe gluster1
```

Do a client mount of the GlusterFS volume. This is needed in `Node1`:

```
sudo mkdir /storage-pool
sudo mount -t glusterfs gluster2:/volume1 /storage-pool
```

Now, let's look at the status of the GlusterFS cluster and created volume in `Node1`:

```
smakam14@sreeubuntu14-VirtualBox1:~$ sudo gluster peer status
Number of Peers: 1

Hostname: gluster2
Uuid: dee83671-cc9d-4037-9ddd-f71e5e4161a0
State: Peer in Cluster (Connected)
```

Now, let's look at `Node2`:

```
smakam14@jungle1:/var/nfs$ sudo gluster peer status
Number of Peers: 1

Hostname: gluster1
Uuid: f3e3141f-5265-4f1e-ba73-7f01b97a633d
State: Peer in Cluster (Connected)
```

Let's look at the volume detail. As we can see, `volume1` is set up as the replicated volume type with two bricks on `gluster1` and `gluster2`:

```
smakam14@sreeubuntu14-VirtualBox1:~$ sudo gluster volume info

Volume Name: volume1
Type: Replicate
Volume ID: e9b66f0e-1d48-4e0d-8b9d-c10e229164e4
Status: Started
Number of Bricks: 1 x 2 = 2
Transport-type: tcp
Bricks:
Brick1: gluster1:/gluster-storage
Brick2: gluster2:/gluster-storage
Options Reconfigured:
nfs.disable: off
```

The following output shows the client mount point in the `df -k` output:

```
gluster1:/volume1  11962496  10353664    978176  92% /storage-pool
```

At this point, we can write and read contents from the client mount point, `/storage-pool`.

Setting up GlusterFS for a CoreOS cluster

By setting up CoreOS nodes to use the GlusterFS filesystem, Container volumes can use GlusterFS to store volume-related data. This allows Containers to move across nodes and keep the volume persistent. CoreOS does not support a local GlusterFS client at this point. One way to use GlusterFS in CoreOS is to export the GlusterFS volume through NFS and do NFS mounting from the CoreOS node.

Continuing to use the GlusterFS cluster created in the previous section, we can enable NFS in the GlusterFS cluster as follows:

```
sudo gluster volume set volume1 nfs.disable off
```

The `cloud-config` for CoreOS that was used in the *Mounting NFS Storage* section can be used here as well. The following is the mount-specific section where we have mounted the GlusterFS volume, `172.17.8.111:/volume1`, in `/mnt/data` of the CoreOS node:

```
- name: mnt-data.mount
  command: start
  content: |
    [Mount]
    What=172.17.8.111:/volume1
    Where=/mnt/data
    Type=nfs
    Options=vers=3,sec=sys,noauto
```

I created a bunch of files in the GlusterFS volume, `/volume1`, and I was able to read and write from the CoreOS node. The following output shows you the `/mnt/data` content in the CoreOS node:

```
core@core-01 /mnt/data $ ls
dump.rdb  file10  file12  file14  file16  file18  file2   file21  file4  file6  file8
file1     file11  file13  file15  file17  file19  file20  file3   file5  file7  file9
```

Accessing GlusterFS using the Docker Volume plugin

Using the GlusterFS volume plugin (`https://github.com/calavera/docker-volume-glusterfs`) for Docker, we can create and manage volumes using a regular Docker volume CLI.

In the following example, we will install the GlusterFS Docker volume plugin and create a persistent `hellocounter` application. I used the same Ubuntu 14.04 VM that is running GlusterFS volumes to run Docker as well.

The following are the steps needed to set up the Docker volume plugin:

- The Docker experimental release supports the GlusterFS volume plugin, so the experimental Docker release needs to be downloaded.
- The GlusterFS Docker volume plugin needs to be downloaded and started. GO (`https://golang.org/doc/install`) needs to be installed to get the volume plugin.
- Use Docker with the GlusterFS Docker volume plugin. For this, the Docker service needs to be stopped and restarted.

The following is the Docker experimental release version running in both the nodes:

```
smakam14@sreeubuntu14-VirtualBox1:~$ docker --version
Docker version 1.9.0-dev, build ccf5b60, experimental
```

Download and start the GlusterFS volume plugin:

```
go get github.com/calavera/docker-volume-glusterfs
sudo docker-volume-glusterfs -servers gluster1:gluster2 &
```

Start the `redis` container with the GlusterFS volume driver as follows:

```
docker run -d -v volume1:/data --volume-driver=glusterfs --name=redis
redis:latest
```

Start the `hellocounter` container and link it to the `redis` container:

```
docker run -d -e USE_REDIS_HOST=redis --link redis:redis -p 80:5000
--name=hellocounter smakam/hellocounter
```

Update the counter by accessing it a few times, as shown in the following screenshot:

```
smakam14@jungle1:/gluster-storage$ cu
Hello World! I have been seen 2 times
```

Now, stop the containers in `node1` and start them in `node2`. Let's see the running containers in `node2`:

```
smakam14@sreeubuntu14-VirtualBox1:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
        NAMES
de0342841f69        smakam/hellocounter   "/bin/sh -c 'python a"   5 hours ago         Up 5 hours          0.0.0.0:5000->50
00/tcp   hellocounter
e85e982828a2        redis:latest          "/entrypoint.sh redis"   5 hours ago         Up 5 hours          6379/tcp
        redis
```

If we access the `hellocounter` container now, we can see that the counter starts from `3` as the previous count is persisted:

```
smakam14@sreeubuntu14-VirtualBox1:~$
Hello World! I have been seen 3 times
```

Ceph

Ceph provides you with distributed storage like GlusterFS and is an open source project. Ceph was originally developed by Inktank and later acquired by Red Hat. The following are some properties of Ceph:

- Ceph uses **Reliable Autonomic Distributed Object Store** (**RADOS**) as the storage mechanism. Other storage access mechanisms such as file and block are implemented on top of RADOS.
- Both Ceph and GlusterFS seem to have similar properties. According to Red Hat, Ceph is positioned more for OpenStack integration and GlusterFS is for Big data analytics, and there will be some overlap.
- There are two key components in Ceph. They are Monitor and OSD. Monitor stores the cluster map and **Object Storage Daemons** (**OSD**) are the individual storage nodes that form the storage cluster. Both storage clients and OSDs use the CRUSH algorithm to distribute the data across the cluster.

Compared to GlusterFS, setting up Ceph seemed a little complex and there is active work going on to run Ceph components as Docker containers as well as integrate Ceph with CoreOS. There is also work going on for the Ceph Docker volume plugin.

NFS

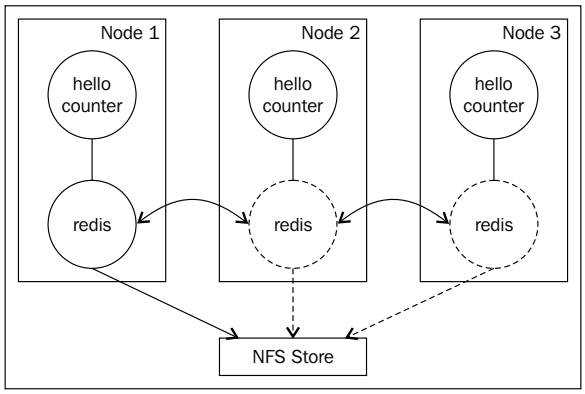
NFS is a distributed filesystem that allows client computers to access network storage as if the storage is attached locally. We can achieve Container data persistence using shared NFS storage.

Container data persistence using NFS

In this section, we will cover a web application example that uses NFS for data persistence. The following are some details of the application:

- The `hellocounter.service` unit starts a container that keeps track of the number of web accesses to the application
- `Hellocounter.service` uses the `redis.service` container to keep track of the access count
- The Redis container uses NFS storage to store the data
- When the database container dies for some reason, Fleet restarts the container in another node in the cluster, and as the service uses NFS storage, the count is persisted

The following figure shows you the example used in this section:



The following are the prerequisites and the required steps:

1. Start the NFS server and a three-node CoreOS cluster mounting the NFS data as specified in the *Mounting NFS storage* section.
2. Start `hellocounter.service` and `redis.service` using fleet with the X-fleet property to control the scheduling of the Containers. The `hellocounter.service` is started on all the nodes; `redis.service` is started on one of the nodes.

The code for `Hellocounter@.service` is as follows:

```
[Unit]
Description=hello counter with redis backend

[Service]
Restart=always
RestartSec=15
ExecStartPre=-/usr/bin/docker kill %p%i
ExecStartPre=-/usr/bin/docker rm %p%i
ExecStartPre=/usr/bin/docker pull smakam/hellocounter

ExecStart=/usr/bin/docker run --name %p%i -e SERVICE_NAME=redis -p
5000:5000 smakam/hellocounter python
app.py

ExecStop=/usr/bin/docker stop %p%i

[X-Fleet]
X-Conflicts=%p@*.service
```

The code for `Redis.service` is as follows:

```
[Unit]
Description=app-redis

[Service]
Restart=always
RestartSec=5
ExecStartPre=-/usr/bin/docker kill %p
ExecStartPre=-/usr/bin/docker rm %p
ExecStartPre=/usr/bin/docker pull redis
ExecStart=/usr/bin/docker run --name redis -v /mnt/data/hellodata:/
data redis

ExecStop=/usr/bin/docker stop %p

[X-Fleet]
Conflicts=redis.service
```

Let's start three instances of `hellocounter@.service` and one instance of `redis.service`. The following screenshot shows three instances of `hellocounter` service and 1 instance of `redis` service running in the CoreOS cluster.

```
core@core-01 ~ $ fleetctl list-units
UNIT                    MACHINE                         ACTIVE  SUB
hellocounter@1.service  8b7a4446.../172.17.8.101         active  running
hellocounter@2.service  51ae77dc.../172.17.8.103         active  running
hellocounter@3.service  3044f610.../172.17.8.102         active  running
redis.service           51ae77dc.../172.17.8.103         active  running
```

As we can see in the preceding screenshot, `hellocounter@2.service` and `redis.service` are in the same node node3.

Let's try accessing the web service from `node3` a few times to check the count:

```
core@core-03 ~ $ curl 172.17.8.103:5000
Hello World! I have been seen 6 times.c
```

The counter value is currently at `6` and stored in NFS.

Now, let's reboot `node3`. As shown in the following output, we can see only two machines:

```
core@core-01 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
3044f610...     172.17.8.102    -
8b7a4446...     172.17.8.101    -
```

Let's look at the services running. As can be seen from the following output, `redis.service` has moved from `node3` to `node2`:

```
core@core-01 ~ $ fleetctl list-units
UNIT                    MACHINE                         ACTIVE  SUB
hellocounter@1.service  8b7a4446.../172.17.8.101        active  running
hellocounter@3.service  3044f610.../172.17.8.102        active  running
redis.service           3044f610.../172.17.8.102        active  running
```

Now, let's check the web access count in `node2`. As we can see from the following output, the count started at `7` as the previous count was set to `6` on `node3`. This proves that container data is persisted:

```
core@core-02 ~ $ curl 172.17.8.102:5000
Hello World! I have been seen 7 times.c
```

Note: This example is not practical as there are multiple instances of a web server operating independently. In a more practical example, a load balancer would be the frontend. This example's purpose is just to illustrate container data persistence using NFS.

The Docker 1.9 update

Docker 1.9 added named volumes, and this makes volumes as a first-class citizen in Docker. Docker volumes can be managed using `docker volume`.

The following screenshot shows you the options in `docker volume`:

```
smakam14@jungle1:~$ docker volume --help

Usage:  docker volume [OPTIONS] [COMMAND]

Manage Docker volumes

Commands:
  create                Create a volume
  inspect               Return low-level information on a volume
  ls                    List volumes
  rm                    Remove a volume
```

A named volume deprecates a data-only container that was used earlier to share volumes across Containers.

The following set of commands shows the same example used earlier with a named volume instead of a data-only container:

```
docker volume create --name redisvolume
docker run -d --name redis1 -v redisvolume:/data redis
docker run -d --name hello1 --link redis1:redis -p 5000:5000 smakam/
hellocounter python app.py
```

In the preceding example, we create a named volume, `redisvolume`, which is used in the `redis1` container. The `hellocounter` application links to the `redis1` container.

The following screenshot shows you information about the `redis1` volume:

```
smakam14@jungle1:~$ docker volume inspect redisvolume
[
    {
        "Name": "redisvolume",
        "Driver": "local",
        "Mountpoint": "/var/lib/docker/volumes/redisvolume/_data"
    }
]
```

Another advantage with named volumes is that we don't need to worry about the dangling volume problem that was present before.

Summary

In this chapter, we covered different storage options available for the storing of Container images and Container data in a CoreOS system. Technologies such as Flocker, GlusterFS, NFS, and Docker volumes and their integration with Containers and CoreOS were illustrated with practical examples. Container storage technologies are still evolving and will take some time to mature. There is a general industry trend to move away from expensive SAN technologies toward local and distributed storage. In the next chapter, we will discuss Container runtime Docker and Rkt and how they integrate with CoreOS.

References

- GlusterFS: `http://gluster.readthedocs.org/`
- The GlusterFS Docker volume plugin: `https://github.com/calavera/docker-volume-glusterfs`
- Flocker: `https://docs.clusterhq.com`

- Docker volume plugins: `https://github.com/docker/docker/blob/master/docs/extend/plugins_volume.md`
- The Docker Storage driver: `https://docs.docker.com/engine/userguide/storagedriver/imagesandcontainers/`
- Docker volumes: `https://docs.docker.com/userguide/dockervolumes/`
- Mounting storage on CoreOS: `https://coreos.com/os/docs/latest/mounting-storage.html`
- The Container filesystem: `http://jpetazzo.github.io/assets/2015-03-03-not-so-deep-dive-into-docker-storage-drivers.html#1`
- Ceph: `http://docs.ceph.com/`
- Ceph Docker: `https://github.com/ceph/ceph-docker`

Further reading and tutorials

- Persistent data storage in the CoreOS cluster: `https://gist.github.com/Luzifer/c184b6b04d83e6d6fbe1`
- Creating a GlusterFS cluster: `https://www.digitalocean.com/community/tutorials/how-to-create-a-redundant-storage-pool-using-glusterfs-on-ubuntu-servers`
- GlusterFS Overview: `https://www.youtube.com/watch?v=kvr6p9gSOX0`
- Stateful Containers using Flocker on CoreOS: `http://www.slideshare.net/ClusterHQ/stateful-containers-flocker-on-coreos-54492047`
- Docker Storage webinar: `https://blog.docker.com/2015/12/persistent-storage-docker/`
- The Contiv volume plugin: `https://github.com/contiv/volplugin`
- Ceph RADOS: `http://ceph.com/papers/weil-rados-pdsw07.pdf`

7
Container Integration with CoreOS – Docker and Rkt

Containers have drastically changed application development and deployment, and are the biggest trend in the computer industry currently. We have talked about Containers in almost all the chapters of this book. In this chapter, we will focus on the Container standards, advanced Docker topics, and basics of the Rkt Container runtime and how all these topics integrate with CoreOS. As Docker is pretty mature, we have covered only advanced Docker topics in this chapter. As the Rkt container runtime is still evolving, we have covered the basics of Rkt in this chapter. Even though Docker started as a Container runtime, Docker has evolved into a Container platform providing orchestration, networking, storage, and security solutions around containers.

The following topics will be covered in this chapter:

- Container standards—**App Container** (**appc**) specification , **Open Container Initiative** (**OCI**), Libnetwork, **Container Network Interface** (**CNI**), and **Cloud Native Computing Foundation** (**CNCF**)
- The Docker daemon configuration, Docker registry, Docker image signing, and basic Docker debugging
- Rkt basics and how to use Rkt with image signing, systemd, and Flannel

Container standards

Standards are an important part of any technology. Standards and specifications allow products and technologies from different vendors to interoperate with each other. As developments in the Container space happened very fast in the last 1-2 years, there was limited attention paid to standards and specifications. In the recent past, the industry has been working toward standards for Container runtime, Container networking, and Container orchestration. In majority of these cases, there are runtime implementations that get released along with the specification and this encourages faster adoption. The following are the standards' categories covered in this section:

- Container image and runtime: APPC and OCI
- Container Networking: Libnetwork and CNI
- Container orchestration: CNCF

App container specification

The APPC specification provides you with a standard to describe the container image format, Container image discovery, Container grouping or Pods, and Container execution environment. Different Container runtimes implementing the APPC specification will be interoperable with each other. The APPC specification is primarily driven by CoreOS along with a few other community members. Rkt, Kurma, and Jetpack are examples of Container runtime implementing APPC. The following are some important components of APPC.

The Container image format

This describes the container image layout, manifest file with image details, and image signing.

Application Container Image (**ACI**) is a Container image created according to the APPC specification. For example, the `nginx.aci` image is an ACI for the nginx Container. To understand Container image format, let's look at what is contained within nginx.aci APPC image. The following command extracts the contents of the `nginx.aci` image to the `nginx` directory: (We got the `nginx.aci` image from the `docker2aci` tool that will be covered later in the chapter.)

```
tar -xvf nginx.aci -C nginx
```

The following set of screenshots shows you the base layout and rootfs layout for the `nginx.aci` image:

```
smakam14@jungle1:~/nginx$ ls
manifest  rootfs
```

```
smakam14@jungle1:~/nginx$ cd rootfs/;ls
bin   dev  home  lib64  mnt  proc  run   srv  tmp  var
boot  etc  lib   media  opt  root  sbin  sys  usr
```

The following are some important sections in the `nginx.aci` manifest. The first screenshot shows the container name and version. The second screenshot describes the exposed ports, mountpoints, environment variables, and so on:

```
  "name": "nginx",
  "acVersion": "0.5.1",
  "acKind": "ImageManifest"
```

```
"app": {
  "ports": [
    {
      "socketActivated": false,
      "count": 1,
      "port": 80,
      "protocol": "tcp",
      "name": "80-tcp"
    },
    {
      "socketActivated": false,
      "count": 1,
      "port": 443,
      "protocol": "tcp",
      "name": "443-tcp"
    }
  ],
  "mountPoints": [
    {
      "path": "/var/cache/nginx",
      "name": "volume-var-cache-nginx"
    }
  ],
  "environment": [
    {
      "value": "/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin",
      "name": "PATH"
    },
```

From the preceding screenshot, we can see that nginx ACI image is exposing ports `80` and `443` and it has mount `point /var/cache/nginx`. Container image signing is done using GPG (`https://www.gnupg.org/`). GPG is a public key cryptography implementation that can be used for the encryption of messages as well as image signing using a public and private key pair.

Container image discovery

Container image discovery describes ways to find the location of Container images from image name. Container images use the URL format. Container image discovery describes ways to find the location of Container images from image name. The following is the image format used:

`https://{name}-{version}-{os}-{arch}.{ext}`

Simple discovery

Here, the complete URL is mentioned to retrieve the ACI image. An example is as follows:

`https://github.com/coreos/etcd/releases/download/v2.0.0/etcd-v2.0.0-linux-amd64.aci`

Meta discovery

Here, the image URL and public key is discovered automatically by using the meta tag embedded in the HTTP location. The following example shows you how to retrieve the meta tag and the ACI image from meta tags for the CoreOS etcd container image.

The first step is to retrieve the meta tags. The `https://coreos.com/etcd` location contains the `ac-discovery` meta tag that contains the image location and the `ac-discovery-pubkeys` meta tag that contains the public key.

The link, `https://coreos.com/etcd/`, contains the following meta tags that can be retrieved as an HTTP request:

```
<meta name="ac-discovery" content="coreos.com/etcd https://github.com/
coreos/etcd/releases/download/{version}/etcd-{version}-{os}-{arch}.
{ext}">
<meta name="ac-discovery-pubkeys" content="coreos.com/etcd https://
coreos.com/dist/pubkeys/aci-pubkeys.gpg">
```

Using the preceding meta tag content, the Container image can be retrieved from the following:

```
https://github.com/coreos/etcd/releases/download/{version}/etcd-
{version}-{os}-{arch}.{ext}
```

The public key can be retrieved from the following:

```
https://coreos.com/dist/pubkeys/aci-pubkeys.gpg
```

The app container executor

The app container executor takes care of the following to set up runtime for the Container:

- UUID setup: This is a Unique ID for the Pod that contains multiple containers. UUID is registered with the metadata service that allows other containers to find each other.
- Filesystem setup: A filesystem is created in its own namespace.
- Volume setup: These are files to be mounted to the container.
- Networking: This specifies Container networking to the host and other Containers.
- Isolators: This controls the CPU and memory limit for the Container.

App container pods

The concept of pods comes from Kubernetes where related containers are packed together in a Pod. Containers within a pod share the process PID, network, and IPC namespace. A manifest can be created for the Pod in addition to individual containers in order to describe properties for the Pod.

The app container metadata service

The app container metadata service is a service that runs externally, and container pods can register information about pods and applications. This metadata service can be used by pods to find information about other pods as well as by containers within a pod to find information about other containers.

APPC tools

APPC provides you with tools to create, validate, and convert ACI images.

Actool

Using Actool for ACI validation:

The following output shows you that the generated ACI image, `busybox-latest.aci`, is a valid APPC image:

```
smakam14@jungle1:~$ actool --debug validate busybox-latest.aci
busybox-latest.aci: valid app container image
```

Using Actool for ACI discovery:

The following output shows you the discovery URL and public key from the ACI image:

```
smakam14@jungle1:~$ actool discover coreos.com/etcd
ACI: https://github.com/coreos/etcd/releases/download/latest/etcd-latest-linux-amd64.aci, ASC: https://github.com/coreos/etcd/releases/download/latest/etcd-latest-linux-amd64.
aci.asc
Keys: https://coreos.com/dist/pubkeys/aci-pubkeys.gpg
```

Using Actool for checking manifest:

The following output shows you how to see the manifest from the ACI image:

```
smakam14@jungle1:~$ actool cat-manifest busybox-latest.aci  | jq .
{
  "annotations": [
    {
      "value": "2015-10-31T22:22:55Z",
      "name": "created"
```

Acbuild

The Acbuild tool is used to build ACI images. The concept is similar to the Dockerfile approach to build Docker Container images, but Acbuild provides more flexibility to build Container images by having better integration with Linux tools such as makefile, environment variables, and others.

The following is an example of building a container image from a GO executable `hello`. Before running the following commands, we need to link the `hello` executable in the current directory statically:

```
acbuild begin
acbuild set-name example.com/hello
acbuild copy hello /bin/hello
acbuild set-exec /bin/hello
acbuild port add www tcp 5000
acbuild label add version 0.0.1
acbuild annotation add authors "Sreenivas Makam<sxxxm@yahoo.com>"
acbuild write hello-0.0.1-linux-amd64.aci
acbuild end
```

If we run the preceding commands, it will create an APPC image, `hello-0.0.1-linux-amd64.aci`, which we can run with the Rkt Container runtime.

The following is another example that is similar to the Dockerfile approach to build an ACI image. In this example, we take a base Ubuntu image, install Apache, and start Apache in a container to create the `ubuntu-nginx.aci` image:

```
acbuild begin
acbuild dependency add quay.io/fermayo/ubuntu
acbuild run -- apt-get update
acbuild run -- apt-get -y install nginx
acbuild set-exec -- /usr/sbin/nginx -g "daemon=off;"
acbuild set-name example.com/ubuntu-nginx
acbuild write ubuntu-nginx.aci
acbuild end
```

To run acbuild, it's necessary to have systemd-nspawn in the system. This is present by default in CoreOS nodes. The following is the APPC image that was created from the preceding script:

```
core@core-01 ~ $ ls -la ubuntu-nginx.aci
-rw-r--r-- 1 root root 29540091 Nov  8 05:23 ubuntu-nginx.aci
```

Let's start the Container using Rkt:

```
core@core-01 ~ $ sudo rkt run --insecure-skip-verify ubuntu-nginx.aci
rkt: using image from local store for image name coreos.com/rkt/stage1-coreos:0.9.0
rkt: using image from file /home/core/ubuntu-nginx.aci
rkt: using image from local store for image name quay.io/fermayo/ubuntu
2015/11/08 06:51:48 Preparing stage1
2015/11/08 06:51:48 Writing image manifest
2015/11/08 06:51:48 Loading image sha512-458e3d17d09add9aec03932415193722718b7f5f07f810dc8dc9c6bc11cf4901
2015/11/08 06:51:49 Writing image manifest
2015/11/08 06:51:49 Writing pod manifest
2015/11/08 06:51:49 Setting up stage1
2015/11/08 06:51:49 Wrote filesystem to /var/lib/rkt/pods/run/da0ca2d9-9b22-4b8e-8837-877c2cab3cbf
2015/11/08 06:51:49 Pivoting to filesystem /var/lib/rkt/pods/run/da0ca2d9-9b22-4b8e-8837-877c2cab3cbf
2015/11/08 06:51:49 Execing /init
```

The following is the running Container's status:

```
core@core-01 ~ $ sudo rkt list pods
UUID            APP             IMAGE NAME                      STATE   NETWORKS
d4285da3        ubuntu-nginx    example.com/ubuntu-nginx        running default:ip4=172.16.28.4
```

Docker2aci

The Docker2aci utility is used to convert Docker Containers to the ACI format. The following is an example that takes a docker `busybox` container and converts it to a `busybox.aci` image:

```
smakam14@jungle1:~$ docker2aci docker://busybox
Downloading d1592a710ac3: [==================================] 674 KB/674 KB
Downloading 17583c7dd0da: [==================================] 32 B/32 B

Generated ACI(s):
busybox-latest.aci
```

Open Container Initiative

OCI is the Open Container Initiative open source project started in April 2015 by Docker and has members from all major companies including Docker and CoreOS. OCI defines the following:

- The Container image format: This describes the filesystem bundle along with `config.json` that describes the host-independent property of a container and `runtime.json` that describes the host-dependent property of a container.
- Runtime: This describes how a container can be started and stopped using namespaces and cgroups.

Docker's goal is to follow the OCI specification for its Container runtime.

Runc

Runc is an implementation of the OCI specification. Docker engine uses runc to implement Container runtime in Docker. Runc can be installed using the procedure described at `https://github.com/opencontainers/runc`.

The following procedure can be used to start a Ubuntu container using `runc`:

```
Docker pull Ubuntu
docker export $(docker create ubuntu) > ubuntu.tar
mkdir rootfs
tar -C rootfs -xf ubuntu.tar
runc spec
```

The first step pulls the Ubuntu Docker Container. The second step exports the Ubuntu Container to a filesystem. The third and fourth steps put the Ubuntu filesystem content in the `rootfs` directory. The last step generates `config.json` and `runtime.json`.

The following output shows you the Ubuntu container started using `runc`:

```
smakam14@jungle1:~/runc$ sudo runc start
# pwd
/
# ls
bin  boot  dev  etc  home  lib  lib64  media  mnt  opt  proc  root  run  sbin  srv  sys  tmp  usr  var
```

The relationship of OCI with APPC

CoreOS, along with a few other community members, created the APPC specification to standardize the Container image format that makes Containers interoperable between different implementations.

The original APPC container specification proposed by CoreOS covers four different elements of container management: packaging, signing, naming (sharing the container with others), and runtime. Docker felt the same need for interoperability and created OCI along with other community members including CoreOS. OCI focuses only on packaging and runtime currently, though this might change in the future. The goals of APPC and OCI are common even though specifics slightly differ. It is possible that these two standards will converge into one standard at some later point.

OCI and APPC latest updates

As per the latest CoreOS blog update (`https://coreos.com/blog/making-sense-of-standards.html`), APPC and OCI will intersect only in runtime and APPC will continue to focus on image format, signing, and distribution.

Libnetwork

Libnetwork was covered briefly in *Chapter 5*, *CoreOS Networking and Flannel Internals*. Libnetwork is an open source project started by Docker and a few other community members with the following objectives:

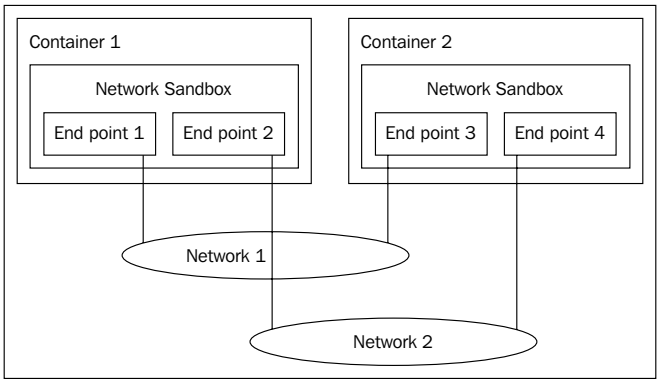
- Keep networking as a library separate from the Container runtime.
- Provide Container connectivity in the same host as well as across hosts.
- Networking implementation will be done as a plugin implemented by drivers. The plugin mechanism is provided to add new third-party drivers easily.
- Control IP address assignment for the Containers using local IPAM drivers and plugins.

Docker uses Libnetwork to provide Container networking.

There are three primary components in Libnetwork:

- **Sandbox**: All networking functionality is encapsulated in a sandbox. This can be implemented using networking namespace or a similar function.
- **Endpoint**: This attaches sandbox to the network.
- **Network**: Multiple endpoints in the same network can talk to each other.

The following diagram shows **Sandbox**, **Endpoint**, and **Network** and how two containers can talk to each other using these constructs:



Libnetwork supports local drivers such as null, bridge, and overlay. The bridge driver can be used for Container connectivity in a single host, and the overlay driver can be used for Container connectivity across hosts. Remote drivers such as Weave and Calico are also supported.

CNI

CNI was covered briefly in *Chapter 5*, *CoreOS Networking and Flannel Internals*.

CNI is the Container networking interface open source project developed by CoreOS along with a few other community members to provide networking facility for Containers as a pluggable and extensible mechanism. CoreOS's Container runtime, Rkt, uses CNI to establish Container networking. The objectives of Libnetwork and CNI are pretty much the same.

The following are some notes on CNI:

- The CNI interface calls the API of the CNI plugin to set up Container networking.
- The CNI plugin is responsible for creating the network interface to the container.

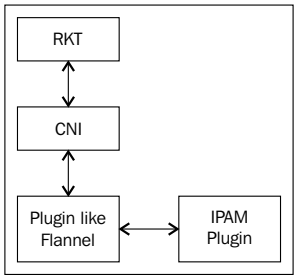
- The CNI plugin calls the IPAM plugin to set up the IP address for the container.
- The CNI plugin needs to implement an API for container network creation and deletion.
- The plugin type and parameters are specified as a JSON file that the Container runtime reads and sets up.
- Available CNI plugins are Bridge, macvlan, ipvlan, and ptp. Available IPAM plugins are host-local and DHCP. CNI plugins and the IPAM plugin can be used in any combination.
- External CNI plugins such as Flannel and Weave are also supported. External plugins reuse the bridge plugin to set up the final networking.
- The following is a sample JSON configuration with the bridge CNI plugin and host-local IPAM plugin along with the IP allocation range:

```
{
    "name": "mynet",
    "type": "bridge",
    "bridge": "mynet0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.10.0.0/16"
    }
}
```

- The following is a sample JSON configuration that uses the Flannel CNI type:

```
{
        "name": "containernet",
        "type": "flannel"
}
```

The following figure shows you the relationship between Rkt, CNI, the CNI plugin, and IPAM plugin:



The relationship between Libnetwork and CNI

Libnetwork and CNI have similar objectives. Docker uses Libnetwork and CoreOS, with Rkt, uses CNI. Libnetwork's overlay driver does something that is similar to CNI's flannel driver. The goal of external plugins such as Weave and Calico is to work with both Libnetwork and CNI.

Cloud Native Computing Foundation

The goal of CNCF is to make it easier to build Cloud native applications using Containers. CNCF will create reference architectures using best open source technologies around Containers for microservice based distributed application. The initial goal of CNCF is Container orchestration and the integration work is focused on Kubernetes with Mesos. CNCF will create the reference architecture for microservice development that can help enterprises to build on the reference architecture rather than integrating components by themselves. As per the latest CoreOS blog (`https://coreos.com/blog/making-sense-of-standards.html`), CoreOS will be donating etcd, flannel, and appc to CNCF.

Docker

Even though Container technology has been available for a long time, Docker has revolutionized the Container technology by making the creation and transportation of Containers very user-friendly. Other than providing Container runtime, Docker provides you with networking, storage, and orchestration solutions for containers. For the majority of these solutions, Docker provides a pluggable model where the Docker native solution is provided, which can be swapped with any other third-party solution. This gives flexibility to the customer to use technologies that they are already comfortable with.

In *Chapter 1*, *CoreOS Overview*, we covered the Docker architecture. As Docker technology is pretty mature, we will cover only the advanced Docker concepts in this chapter.

The Docker daemon and an external connection

Docker runs as a daemon and by default listens on the Unix socket, `unix:///var/run/docker.sock`. Docker start options are specified in `/etc/default/docker`.

To allow external Docker clients to talk to the Docker daemon, the following procedure is to be performed in the Ubuntu node:

1. Add the TCP server with the local address and port number:

   ```
   DOCKER_OPTS="-D -H unix:///var/run/docker.sock -H
   tcp://192.168.56.101:2376"
   ```

2. Restart the docker daemon:

   ```
   Sudo service docker restart
   ```

3. Now, we can see that the Docker daemon is exposing external connectivity on the IP address `192.168.56.101` and TCP port number `2376`:

```
root      7676     1  0 14:16 ?        00:00:00 /usr/bin/docker daemon -D -H unix:///var/run/docker.
sock -H tcp://192.168.56.101:2376
```

We can connect from external Docker clients as follows:

```
docker -H tcp://192.168.56.101:2376 ps
```

The following image shows that apache container is running:

```
$ docker -H tcp://192.168.56.101:2376 ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                   NAMES
a87aee202a47        smakam/apache1      "/usr/sbin/apache2ctl"   4 minutes ago       Up 4 minutes        0.0.0.0:8080->80/tcp    apache
```

Dockerfile

Dockerfile is used to create Docker Container images using specified instructions in Dockerfile. Typically, Dockerfile starts with a base container image, installs the necessary applications, and starts the process associated with the container.

For Dockerfile best practices, you can refer to the following link:

`https://docs.docker.com/engine/articles/dockerfile_best-practices/`

The following is an example Dockerfile for creating an Apache container from the Ubuntu base image. This Dockerfile installs the Apache package and exposes port `80` to the outside world from the Container:

```
FROM ubuntu:14.04
MAINTAINER Sreenivas Makam <sxxxm@yahoo.com>
# Update
RUN apt-get update
# Install apache2
RUN apt-get install -y apache2
# Expose necessary ports
EXPOSE 80
```

```
# Start application
ENTRYPOINT ["/usr/sbin/apache2ctl"]
CMD ["-D", "FOREGROUND"]
```

To create a Docker image, execute the following command in the directory where the preceding Dockerfile is present. In the example below, `smakam/apache1` is the name of the Container image. The default convention for Container image name is `username/imagename:tag`.

```
docker build -t smakam/apache1 .
```

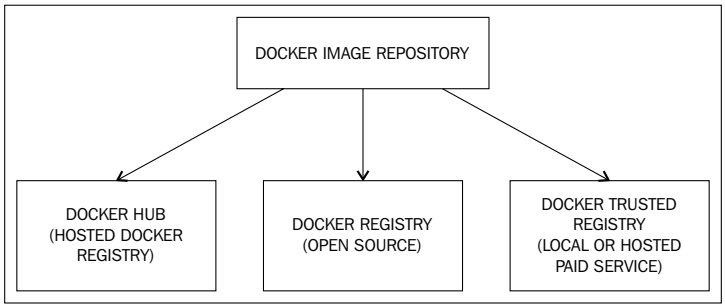
The following screenshot shows you the created Apache container image:

The Docker Image repository

The Docker image repository is used to save and restore Docker Container images from a common server location. There are three possible solutions that Docker provides for storing Container images:

- **Docker hub**: This is the Docker image repository service that's hosted by Docker itself at `https://hub.docker.com/`. This is a free service provided by Docker.
- **Docker registry**: This is an open source project (`https://github.com/docker/distribution`) that allows customers to host the Docker registry in their own premises. The latest Docker registry is version 2.0. Docker registry 2.0 overcomes some of the shortcomings of Docker registry 1.*x* for better security and performance.
- **Docker Trusted registry**: This is Docker's commercial implementation (`https://www.docker.com/docker-trusted-registry`) of the Docker registry and adds features such as role-based user authentication, integration with an external directory service such as LDAP, GUI-based administrative management, support, and so on. Both the Docker registry and Docker Trusted registry support integration with external storage drivers such as AWS, Azure, and Swift to store Docker images.

The following diagram captures the three Docker image repository types:



Docker images have this format:

```
[REGISTRYHOST/][USERNAME/]NAME[:TAG]
```

- `REGISTRYHOST`: The registry server address
- `USERNAME`: The username that created the image
- `NAME`: The Container image name
- `TAG`: The version of the Container image
- Except `NAME`, the other arguments are optional

For example, the following command will pull a standard Ubuntu container image from the Docker hub; `registry-1.docker.io/library` is the registry host, the name is `Ubuntu`, and the tag is `latest`:

```
docker pull registry-1.docker.io/library/Ubuntu:latest
```

Similar to the Docker registry, CoreOS has the Quay registry (`https://quay.io/`) to store Docker and Rkt images, and they have a public and enterprise version available.

Creating your own Docker registry

It is useful to create a local registry to share images in a particular company or group. This is important from a security perspective since there is no need to access Internet to access registry. The Docker registry provides you with options for authentication, backend storage drivers (for example, S3, Azure, and Swift), logging, and so on.

To start a local registry, use the following command:

```
docker run -d -p 5000:5000 --restart=always --name registry registry:2
```

The following screenshot shows you the registry running as container. The registry service is exposed on port `5000` in the localhost:

```
smakam14@jungle1:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
     PORTS                    NAMES
10fd98a3c885        registry:2          "/bin/registry /etc/d"   43 hours ago        Up About a minu
te   0.0.0.0:5000->5000/tcp   registry
```

The registry configuration is specified either as an environment variable as part of starting the registry container or using a YAML file with the configuration and mounting this YAML file to `/etc/config/registry/config.yaml` in the container.

The following set of commands pulls a busybox container from the Docker hub, pushes the busybox container in the local registry, and then pulls it out from the local registry:

```
docker pull busybox
docker tag busybox localhost:5000/mybusybox
docker push localhost:5000/mybusybox
docker pull localhost:5000/mybusybox
```

The following screenshot shows you the `mybusybox` container that has been pulled from the local registry:

```
smakam14@jungle1:~$ docker images | grep mybusybox | grep localhost
localhost:5000/mybusybox             latest              c9eda10ebdb4        6 days ago          1.
113 MB
```

The following screenshot shows you the instantiation of the `mybusybox` container from the local registry:

```
smakam14@jungle1:~$ docker run localhost:5000/mybusybox echo "hello"
hello
```

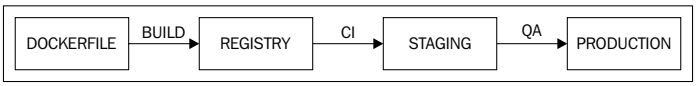
Continuous integration

When we push Docker images to the Docker Hub, Dockerfile does not get pushed. To push the Dockerfile and use it for automated Container builds, we need to link it with a repository management tool such as GitHub or Bitbucket. The steps are as follows:

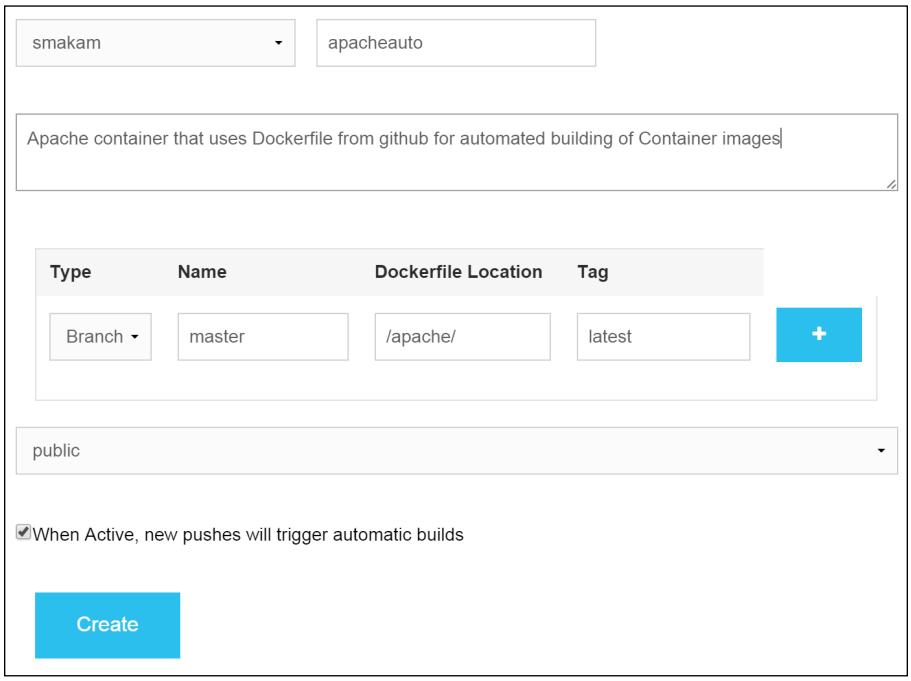
1. Get an account in GitHub or Bitbucket.
2. From Docker Hub, we can link either to GitHub or Bitbucket.
3. Push the Dockerfile to GitHub.

4. From DockerHub, when we create a repository, select automated build, and select the location from GitHub where Dockerfile is present. This will build the image automatically. Additionally, when there are changes to Dockerfile committed to GitHub, automatic builds are triggered.

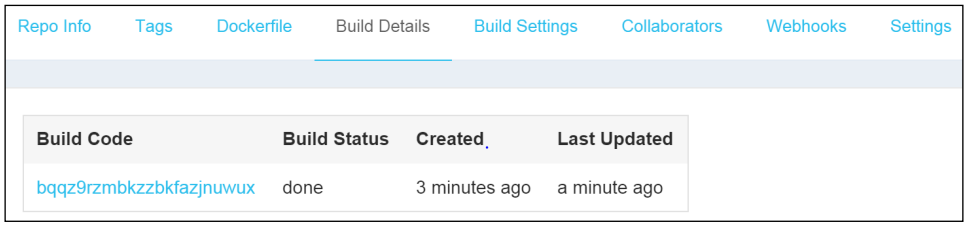
The following diagram shows you the CI sequence using Dockerfile, from staging to production:



The following screenshot shows the automated build creation in Docker Hub after creating Dockerfile in GitHub. In the following example, Dockerfile is present in `https://github.com/smakam/docker.git` under the Apache directory:



When any changes are made, an automatic container image is built. The following screenshot shows the successful container image build log for `smakam/apacheauto`:



The following screenshot shows a successful pull of the `smakam/apacheauto` container image:

```
smakam14@jungle1:~$ docker run -d -p 8080:80 smakam/apacheauto
ab5b1e7b7d563c0891a59460b3ddca74b0bf8a564e1cfc4d0b6a0be47baab749
smakam14@jungle1:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS
     PORTS                  NAMES
ab5b1e7b7d56        smakam/apacheauto   "/usr/sbin/apache2ctl"   4 seconds ago       Up 3 seconds
     0.0.0.0:8080->80/tcp     jovial_hypatia
```

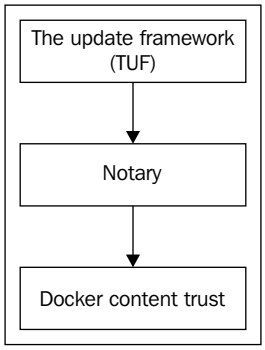
The Docker content trust

The Docker content trust provides you with a mechanism to sign and publish Docker images so that the client who pulls the image can be guaranteed that the image is from a trusted source and has not been modified by any man-in-the-middle attack. The following are some features of the Docker content trust:

- The Docker content trust is an implementation of the Notary open source project (`https://github.com/docker/notary`). The Notary open source project is based on **The Update Framework** (**TUF**) project (`https://theupdateframework.github.io/`). TUF provides a mechanism to secure software updates.
- Compared to the GPG approach of signing keys, TUF has some unique differentiators. TUF takes care of the freshness of keys so that the client always knows that they are getting the latest content. Key compromise can be handled better with TUF using the key rotation scheme, which clients need not be aware of. TUF also provides you with the capability of signing collections rather than individual software.
- There are four keys with Notary—the Timestamp key to maintain the freshness of the image, the Snapshot key to sign image collections, the Target key for the regular signing of images, and the Offline key for key rotation.

- The Docker content trust has been released with Docker version 1.8. The default option is trust-disabled and can be enabled using the `DOCKER_CONTENT_TRUST` environment variable. At some later point, the default option would be to keep the trust enabled.

The following figure shows you the relationship between TUF, Notary, and the Docker content trust:



The following is the workflow with the Docker content trust:

- The Docker registry needs to support the Docker content trust. The Docker Hub supports the content trust. The Docker trusted registry and private registry do not yet support the content trust; this will be added soon.
- The usual Docker commands can be used for push and pull, and care has been taken not to change Docker commands. For advanced key management, the Notary CLI can be used.
- When the publisher pushes the image for the first time using `docker push`, there is a need to enter a passphrase for the root key and tagging key. All other keys are generated automatically. These keys need to be stored safely.
- For any further image publishing, only the tagging key is necessary.
- The client has the option to pull signed or unsigned images. With the Docker trust enabled, the client will get an error if they try to pull unsigned images.

Pushing secure image

First, we enable the Docker content trust using the `DOCKER_CONTENT_TRUST` environment variable. The following is the output when the Docker content trust is enabled and we are publishing the image for the first time. Here, we are pushing the signed `smakam/mybusybox:v1` container:

```
smakam14@jungle1:~$ export DOCKER_CONTENT_TRUST=1
smakam14@jungle1:~$ docker push smakam/mybusybox:v1
The push refers to a repository [docker.io/smakam/mybusybox] (len: 1)
defba46cb616: Pushed
8c2e06607696: Pushed
6ce2e90b0bc7: Pushed
cf2616975b4a: Pushed
v1: digest: sha256:a9ec29301095ac255c44a302c795e27c79e31ad2bc71d68cdcbdba74ebb79f73 size: 5580
Signing and pushing trust metadata
[DEPRECATED] The environment variable DOCKER_CONTENT_TRUST_OFFLINE_PASSPHRASE has been deprecated a
d will be removed in v1.10. Please use DOCKER_CONTENT_TRUST_ROOT_PASSPHRASE
[DEPRECATED] The environment variable DOCKER_CONTENT_TRUST_TAGGING_PASSPHRASE has been deprecated a
d will be removed in v1.10. Please use DOCKER_CONTENT_TRUST_REPOSITORY_PASSPHRASE
You are about to create a new root signing key passphrase. This passphrase
will be used to protect the most sensitive key in your signing system. Please
choose a long, complex passphrase and be careful to keep the password and the
key file itself secure and backed up. It is highly recommended that you use a
password manager to generate the passphrase and keep it safe. There will be no
way to recover this key. You can find the key in your config directory.
Enter passphrase for new root key with id 4c12027:
Repeat passphrase for new root key with id 4c12027:
Enter passphrase for new repository key with id docker.io/smakam/mybusybox (001986b):
Repeat passphrase for new repository key with id docker.io/smakam/mybusybox (001986b):
Finished initializing "docker.io/smakam/mybusybox"
```

Pulling secure image

The following is the output when we are pulling the same secure image, `smakam/mybusybox:v1`, from the Docker hub:

```
smakam14@sreeubuntu14-VirtualBox1:~$ export DOCKER_CONTENT_TRUST=1
smakam14@sreeubuntu14-VirtualBox1:~$ docker pull smakam/mybusybox
Using default tag: latest
[DEPRECATED] The environment variable DOCKER_CONTENT_TRUST_OFFLINE_PASSPHRASE has been deprecated an
d will be removed in v1.10. Please use DOCKER_CONTENT_TRUST_ROOT_PASSPHRASE
[DEPRECATED] The environment variable DOCKER_CONTENT_TRUST_TAGGING_PASSPHRASE has been deprecated an
d will be removed in v1.10. Please use DOCKER_CONTENT_TRUST_REPOSITORY_PASSPHRASE
No trust data for latest
smakam14@sreeubuntu14-VirtualBox1:~$ docker pull smakam/mybusybox:v1
[DEPRECATED] The environment variable DOCKER_CONTENT_TRUST_OFFLINE_PASSPHRASE has been deprecated an
d will be removed in v1.10. Please use DOCKER_CONTENT_TRUST_ROOT_PASSPHRASE
[DEPRECATED] The environment variable DOCKER_CONTENT_TRUST_TAGGING_PASSPHRASE has been deprecated an
d will be removed in v1.10. Please use DOCKER_CONTENT_TRUST_REPOSITORY_PASSPHRASE
Pull (1 of 1): smakam/mybusybox:v1@sha256:a9ec29301095ac255c44a302c795e27c79e31ad2bc71d68cdcbdba74eb
b79f73
sha256:a9ec29301095ac255c44a302c795e27c79e31ad2bc71d68cdcbdba74ebb79f73: Pulling from smakam/mybusyb
ox

dd07788819d1: Pull complete
ad74b8b5fc11: Pull complete
f361d8111a5b: Pull complete
3e6f8ae5ca93: Pull complete
Digest: sha256:a9ec29301095ac255c44a302c795e27c79e31ad2bc71d68cdcbdba74ebb79f73
Status: Downloaded newer image for smakam/mybusybox@sha256:a9ec29301095ac255c44a302c795e27c79e31ad2b
c71d68cdcbdba74ebb79f73
Tagging smakam/mybusybox@sha256:a9ec29301095ac255c44a302c795e27c79e31ad2bc71d68cdcbdba74ebb79f73 as
smakam/mybusybox:v1
```

Pulling same image with no security

The following is the output when we try to pull the same image, `smakam/mybusybox:v1`, with no Docker content trust. In this case, image verification is not done:

```
smakam14@sreeubuntu14-VirtualBox1:~$ docker pull smakam/mybusybox:v1
v1: Pulling from smakam/mybusybox

Digest: sha256:a9ec29301095ac255c44a302c795e27c79e31ad2bc71d68cdcbdba74ebb79f73
Status: Downloaded newer image for smakam/mybusybox:v1
```

The following is the error message that we will get if we enable the trust and try to pull Docker images that are not signed. As `smakam/hellocounter` is not signed and we have `DOCKER_CONTENT_TRUST` enabled, we get an error:

```
smakam14@sreeubuntu14-VirtualBox1:~$ docker pull smakam/hellocounter
Using default tag: latest
[DEPRECATED] The environment variable DOCKER_CONTENT_TRUST_OFFLINE_PASSPHRASE has been deprecated an
d will be removed in v1.10. Please use DOCKER_CONTENT_TRUST_ROOT_PASSPHRASE
[DEPRECATED] The environment variable DOCKER_CONTENT_TRUST_TAGGING_PASSPHRASE has been deprecated an
d will be removed in v1.10. Please use DOCKER_CONTENT_TRUST_REPOSITORY_PASSPHRASE
no trust data available
```

Recently, Docker has enabled the content trust using hardware keys (`https://blog.docker.com/2015/11/docker-content-trust-yubikey/`). This is currently in the experimental mode.

Container debugging

The following are some basic Container debugging approaches.

Logs

The following command will show container logs. This can be a useful debugging tool. In *Chapter 10, CoreOS and Containers - Troubleshooting and Debugging*, you will learn how to aggregate and analyze Container logs from a central location.

```
docker logs <containername or id>
```

Login inside Container

The `docker exec` command can be used to log in to the container. The following is an example:

```
smakam14@jungle1:~/docker/apache$ docker exec -ti apache sh
# 
```

Common Linux commands can be executed from the Container shell.

Container properties

The following command will show container properties such as mount points, resource limits, and so on:

```
docker inspect <containername or id>
```

Container processes

The following command will show processes running in the container sorted by the process CPU usage:

```
docker top <containername or id>
```

The following is a sample output for a `redis` container:

```
core@core-01 ~ $ docker top redis
UID                 PID                 PPID                C                   STIME               TTY                 TIME                CMD
999                 1688                699                 0                   05:26               ?                   00:00:09            redis-server *:6379
```

The Container's CPU and memory usage

The following command will show the resource usage of a Container:

```
docker stats <containername or id>
```

The following is a sample output for the Apache container:

```
CONTAINER           CPU %               MEM USAGE / LIMIT     MEM %               NET I/O             BLOCK I/O
apache              0.12%               6.795 MB / 6.382 GB   0.11%               4.292 kB / 648 B    0 B / 0 B
```

Rkt

Rkt is the Container runtime from CoreOS based on the APPC specification.
The following are some differences in Rkt compared to Docker:

- Rkt is daemonless. The problem of Containers going away if the Docker daemon restarts does not exist with Rkt.
- Rkt integrates well with systemd so that container resource limits can be set easily for the Containers.

There are three stages in the Rkt execution:

- `Stage0`: This does the image discovery and retrieval and sets up a filesystem for stages 1 and 2.

- `Stage1`: This sets up the execution environment for the container execution using the filesystem set up by `stage0`. Rkt uses systemd-nspawn to set up cgroups, networking, and so on in this stage. The goal here is to keep `stage1` swappable by other implementations.
- `Stage2`: This is the actual execution of the Container pod and application itself using the execution environment set up by `stage1` and filesystem set up by `stage0`.

The following example illustrates the stages. Let's start the `hello` ACI image using Rkt:

```
sudo rkt --insecure-skip-verify run hello-0.0.1-linux-amd64.aci
```

```
smakam14@jungle1:~/rkt$ sudo rkt run --insecure-skip-verify hello-0.0.1-linux-amd64.aci
rkt: using image from file /home/smakam14/rkt-v0.10.0/stage1-coreos.aci
rkt: using image from file /home/smakam14/rkt/hello-0.0.1-linux-amd64.aci
run: group "rkt" not found, will use default gid when rendering images
```

The following shows the `stage1` filesystem setup by `stage0`:

```
root@jungle1:/var/lib/rkt/pods/run/df40b3d4-e807-4127-add8-08e32a4eaa9d/stage1# ls
manifest  rootfs
```

The manifest here shows the Rkt `stage1` ACI that sets up the container environment:

```
  "name": "coreos.com/rkt/stage1-kvm",
  "acVersion": "0.7.1",
```

The following shows the `stage2` filesystem:

```
root@jungle1:/var/lib/rkt/pods/run/df40b3d4-e807-4127-add8-08e32a4eaa9d/stage1/rootfs/opt/stage2/hello# ls
manifest  rootfs
```

The manifest here shows the hello Rkt container image:

```
  "name": "example.com/hello",
  "acVersion": "0.7.1+git",
```

The following shows the filesystem for the hello application:

```
root@jungle1:/var/lib/rkt/pods/run/df40b3d4-e807-4127-add8-08e32a4eaa9d/stage1/rootfs/opt/stage2/hello/rootfs# ls
bin  dev  etc  proc  sys  tmp
```

Rkt application is available in the CoreOS base image. Rkt can also be installed in any Linux system using the procedure described at `https://github.com/coreos/rkt`. The following is the Rkt version running in the Ubuntu 14.04 system:

```
smakam14@jungle1:~$ rkt version
rkt version 0.10.0
appc version 0.7.1
```

The following is the Rkt and APPC version used in the CoreOS alpha image 815.0.0:

```
core@core-01 ~ $ rkt version
rkt version 0.8.1
appc version 0.6.1
```

Basic commands

The following are some basic commands to manipulate Rkt Containers.

Fetch image

The following command fetches a Container image from the repository in the ACI format:

```
sudo rkt --insecure-skip-verify fetch docker://busybox
```

```
core@core-01 ~ $ sudo rkt --insecure-skip-verify fetch docker://busybox
Downloading d1592a710ac3: [=====================================] 674 KB/674 KB
Downloading 17583c7dd0da: [=====================================] 32 B/32 B
sha512-cf74c26d8d35555066dce70bd94f513b
```

List images

The following command lists Rkt Container images:

```
sudo rkt image list
```

```
core@core-01 ~ $ sudo rkt image list
KEY                                                             NAME                                IMPORTTIME                          LATEST
sha512-0f2cbc023a29060268ada6082593c8fdbc54117f342b84f04294dee595179a08 coreos.com/rkt/stage1:0.8.1    2015-11-06 17:25:26.826 +0000 UTC   false
sha512-cf74c26d8d35555066dce70bd94f513b90cbef6e7e9c01ea0c971f4f6d689848 busybox:latest                2015-11-07 09:30:18.219 +0000 UTC   true
```

Run image

The following command runs the Rkt Container image:

```
sudo rkt run --insecure-skip-verify --interactive docker://busybox
```

By default, signature verification is turned on; we disable signature verification using the `skip-verify` option:

```
core@core-01 ~ $ sudo rkt run --insecure-skip-verify --interactive docker://busybox
Downloading d1592a710ac3: [==================================] 674 KB/674 KB
Downloading 17583c7dd0da: [==================================] 32 B/32 B
2015/11/07 09:34:17 Preparing stage1
2015/11/07 09:34:17 Loading image sha512-cf74c26d8d35555066dce70bd94f513b90cbef6e7e9c01ea0c971f4f6d689848
2015/11/07 09:34:17 Writing pod manifest
2015/11/07 09:34:17 Setting up stage1
2015/11/07 09:34:17 Writing image manifest
2015/11/07 09:34:17 Wrote filesystem to /var/lib/rkt/pods/run/1d15bc7e-9a2e-4f65-9fbf-dafbbaa544e6
2015/11/07 09:34:17 Writing image manifest
2015/11/07 09:34:17 Pivoting to filesystem /var/lib/rkt/pods/run/1d15bc7e-9a2e-4f65-9fbf-dafbbaa544e6
2015/11/07 09:34:17 Execing /init
/ #
```

List pods

The following command lists the running pods:

```
sudo rkt list pods
```

```
core@core-01 ~ $ sudo rkt list pods
UUID            APP     ACI     STATE   NETWORKS
1d15bc7e        busybox busybox running
```

Garbage collection

The following screenshot shows two pods that have exited:

```
core@core-01 ~ $ sudo rkt list pods
UUID            APP     ACI     STATE   NETWORKS
1d15bc7e        busybox busybox exited
1e321f1f        busybox busybox exited
```

The exited Containers will be garbage collected periodically. To force garbage collection, we can perform the following command:

```
rkt gc --grace-period=0
```

```
core@core-01 ~ $ sudo rkt gc --grace-period=0
Moving pod "1d15bc7e-9a2e-4f65-9fbf-dafbbaa544e6" to garbage
Moving pod "1e321f1f-a6f9-422e-b0fe-f7c71da850c2" to garbage
Garbage collecting pod "1d15bc7e-9a2e-4f65-9fbf-dafbbaa544e6"
Garbage collecting pod "1e321f1f-a6f9-422e-b0fe-f7c71da850c2"
```

Now, we can see that there are no active pods:

```
core@core-01 ~ $ sudo rkt list
UUID    APP     ACI     STATE   NETWORKS
core@core-01 ~ $
```

Delete image

The following command deletes the local Container image:

```
sudo rkt image rm sha512-cf74c26d8d35555066dce70bd94f513b90cbef6e7e9c01ea
0c971f4f6d689848
```

The following screenshot shows the deletion of the busybox image using UUID:

```
core@core-01 ~ $ sudo rkt image rm sha512-cf74c26d8d35555066dce70bd94f513b90cbef6e7e9c01ea0c971f4f6d689848
rkt: successfully removed aci for imageID: "sha512-cf74c26d8d35555066dce70bd94f513b90cbef6e7e9c01ea0c971f4f6d689848"
rkt: 1 image(s) successfully removed
```

Export image

The following command converts a Docker image to the ACI format:

```
sudo rkt image export nginx nginx.aci
```

```
core@core-01 ~ $ sudo rkt image export nginx nginx.aci
core@core-01 ~ $ ls
nginx.aci
```

The nginx container with volume mounting and port forwarding

The following command starts the nginx container forwarding the container port `80` to the host port `8080` and setting up the host volume. The volume directory and port name are as specified in the manifest file:

```
sudo rkt run --insecure-skip-verify --private-net --port=80-tcp:8080
--volume volume-var-cache-nginx,kind=host,source=/home/core docker://
nginx
```

```
core@core-01 ~ $ sudo rkt list pods
UUID            APP     ACI     STATE   NETWORKS
2b165196        nginx   nginx   running containernet:ip4=10.1.74.5, default:ip4=172.16.28.9
```

The following screenshot shows successful web page access using the nginx container and host port `8080`:

```
core@core-01 ~ $ curl 172.17.8.101:8080
<!DOCTYPE html>
<html>
<head>
<title>Welcome to nginx!</title>
```

Pod status

The following command lists the status of a particular Pod using UUID:

```
sudo rkt status 2b165196
```

```
core@core-01 ~ $ sudo rkt status 2b165196
state=running
networks=containernet:ip4=10.1.74.5, default:ip4=172.16.28.9
pid=4085
exited=false
```

Rkt image signing

Container image signing allows us to verify that the image is coming from a trusted location and has not been tampered with. I used the procedure at `https://github.com/coreos/rkt/blob/master/Documentation/signing-and-verification-guide.md` to sign the ACI image and use Rkt to run the signed image.

The following is a sample `nginx.service` systemd unit file:

1. Generate keys:

   ```
   gpg --batch --gen-key gpg-batch
   ```

In case you get this error message, `Not enough random bytes available. Please do some other work to give the OS a chance to collect more entropy!`, it can be solved by the following rngd tool that can be run in parallel:

```
apt-get install rng-tools
sudo rngd -r /dev/urandom
```

2. Trust the keys:

   ```
   gpg --no-default-keyring --secret-keyring ./rkt.sec --keyring ./
   rkt.pub --edit-key 1FEEF0ED trust
   ```

3. Export the public key:

```
gpg --no-default-keyring --armor \
--secret-keyring ./rkt.sec --keyring ./rkt.pub \
--export <email> > pubkeys.gpg
```

4. Sign the image using the public key:

```
gpg --no-default-keyring --armor \
--secret-keyring ./rkt.sec --keyring ./rkt.pub \
--output hello-0.0.1-linux-amd64.aci.asc \
--detach-sig hello-0.0.1-linux-amd64.aci
```

5. Host the web server with the ACI image, public key, and signature. The following are the contents in my web server location:

```
smakam14@jungle1:~/rkt/www-data$ ls
hello-0.0.1-linux-amd64.aci  hello-0.0.1-linux-amd64.aci.asc  index.html  pubkeys.gpg
```

6. The following is the `index.html` content:

```
<head>
<meta name="ac-discovery" content="example.com/hello http://
example.com/hello-0.0.1-linux-amd64.aci">
<meta name="ac-discovery-pubkeys" content="example.com/hello
http://example.com/pubkeys.gpg">
</head>
```

7. Trust the web server location and key:

```
sudo rkt trust --prefix=example.com/hello http://example.com/
pubkeys.gpg --insecure-allow-http
```

8. Modify `/etc/hosts` to point `example.com` to localhost.
9. Start a simple web server:

```
sudo python -m SimpleHTTPServer 80
```

Now, we can run the Rkt image with signature verification:

```
sudo rkt run --debug http://example.com/hello-0.0.1-linux-amd64.aci
```

The following screenshot shows the signature being verified. The signature location and public key are provided by the hosted web server at `example.com`:

```
smakam14@jungle1:~/rkt$ sudo rkt run --debug http://example.com/hello-0.0.1-linux-amd64.aci
rkt: using image from file /home/smakam14/rkt-v0.10.0/stage1-coreos.aci
rkt: remote fetching from url http://example.com/hello-0.0.1-linux-amd64.aci
rkt: fetching image from http://example.com/hello-0.0.1-linux-amd64.aci
Downloading signature from http://example.com/hello-0.0.1-linux-amd64.aci.asc
Downloading signature: [======================================] 473 B/473 B
Downloading ACI: [============================================] 1.67 MB/1.67 MB
rkt: signature verified:
  Sreenivas Makam (ACI signing key) <smakam@yahoo.com>
```

Rkt with systemd

Systemd provides you with a lot of control over how processes are managed. Rkt pods can be managed by systemd. With systemd, we can control the process execution order, restartability, resource limit, and so on.

The following is a sample `nginx.service` systemd unit file:

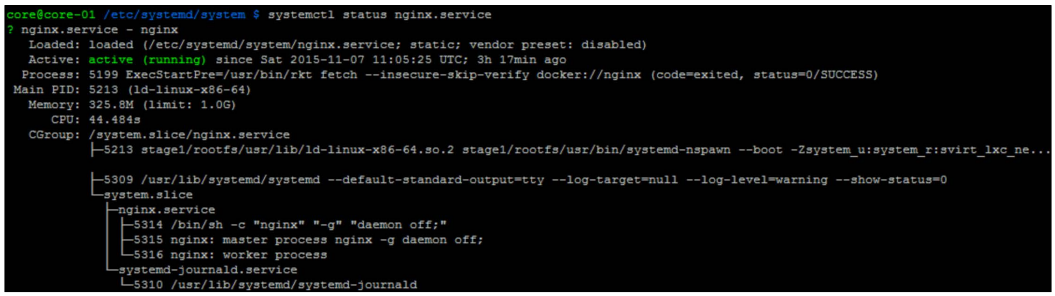
```
[Unit]
Description=nginx

[Service]
# Resource limits
CPUShares=512
MemoryLimit=1G
# Prefetch the image
ExecStartPre=/usr/bin/rkt fetch --insecure-skip-verify docker://nginx
ExecStart=/usr/bin/rkt run --insecure-skip-verify --private-net
--port=80-tcp:8080 --volume volume-var-cache-nginx,kind=host,source=/
home/co
re docker://nginx
KillMode=mixed
Restart=always
```

In the preceding service file, we started the nginx container and also limited the CPU and memory usage for this `nginx.service` using the systemd construct.

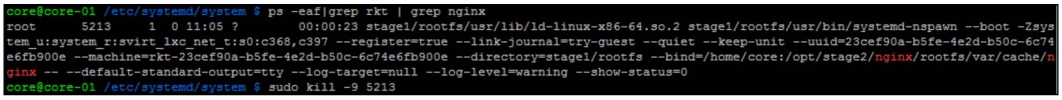
To start the service, it's necessary to place `nginx.service` in `/etc/systemd/system`. The service can be started as follows:

```
Sudo systemctl start nginx.service
```

The following screenshot shows you the status of `nginx.service`:

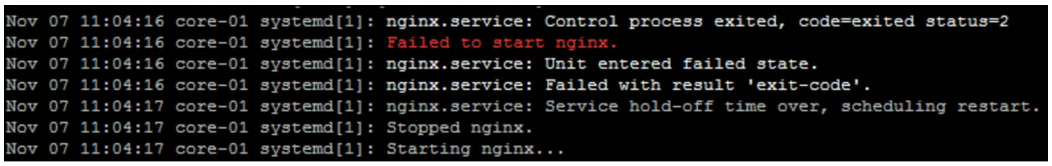


To show the power of integration with systemd, let's kill the Rkt nginx process and demonstrate restartability:

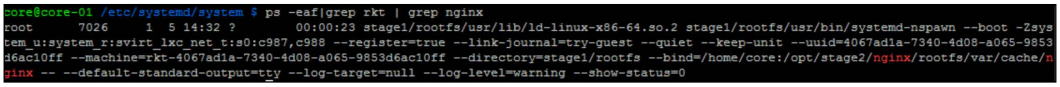


Systemd will restart the nginx container because `restart` is turned on in `nginx.service`.

From the following `journalctl` logs on `nginx.service`, we can see that the service has been restarted:



In the following screenshot, we can see that the Rkt nginx process is running with a different PID:



Rkt with Flannel

Rkt uses the CNI interface to talk to the Flannel plugin to establish container networking across hosts.

The following example sets up a three-node CoreOS cluster using Rkt and Flannel for Container networking. The following is the necessary `cloud-config`:

```
#cloud-config

coreos:
  etcd2:
    #generate a new token for each unique cluster from https://
discovery.etcd.io/new
    discovery: <your token>
    # multi-region and multi-cloud deployments need to use $public_
ipv4
    advertise-client-urls: http://$public_ipv4:2379
    initial-advertise-peer-urls: http://$private_ipv4:2380
    # listen on both the official ports and the legacy ports
    # legacy ports can be omitted if your application doesn't depend
on them
    listen-client-urls: http://0.0.0.0:2379,http://0.0.0.0:4001
    listen-peer-urls: http://$private_ipv4:2380,http://$private_
ipv4:7001
  fleet:
    public-ip: $public_ipv4
  flannel:
    interface: $public_ipv4
  units:
    - name: etcd2.service
      command: start
    - name: fleet.service
      command: start
    - name: flanneld.service
      drop-ins:
        - name: 50-network-config.conf
          content: |
            [Service]
            ExecStartPre=/usr/bin/etcdctl set /coreos.com/network/
config '{ "network": "10.1.0.0/16" }'
      command: start
    - name: docker-tcp.socket
      command: start
      enable: true
      content: |
        [Unit]
        Description=Docker Socket for the API
```

```
            [Socket]
            ListenStream=2375
            Service=docker.service
            BindIPv6Only=both

            [Install]
            WantedBy=sockets.target
    write_files:
      - path: "/etc/rkt/net.d/10-containernet.conf"
        permissions: "0644"
        owner: "root"
        content: |
          {
            "name": "containernet",
            "type": "flannel"
          }
```

The `/etc/rkt/net.d/10-containernet.conf` file sets up the CNI plugin type as Flannel.

Flannel gets an individual subnet for each host using the IP range specified in the flannel configuration `10.1.0.0/16`.

The following output shows you the subnet allocated in `node1` and `node2`:

```
core@core-01 ~ $ cat /run/flannel/subnet.env
FLANNEL_NETWORK=10.1.0.0/16
FLANNEL_SUBNET=10.1.74.1/24
FLANNEL_MTU=1472
```

```
core@core-03 ~ $ cat /run/flannel/subnet.env
FLANNEL_NETWORK=10.1.0.0/16
FLANNEL_SUBNET=10.1.3.1/24
FLANNEL_MTU=1472
```

Let's create Rkt containers in each node and check inter-container connectivity:

```
core@core-01 ~ $ sudo rkt run --private-net --interactive --insecure-skip-verify docker://busybox
Downloading d1592a710ac3: [==================================] 674 KB/674 KB
Downloading 17583c7dd0da: [==================================] 32 B/32 B
2015/11/07 07:39:00 Preparing stage1
2015/11/07 07:39:00 Loading image sha512-cf74c26d8d35555066dce70bd94f513b90cbef6e7e9c01ea0c971f4f6d689848
2015/11/07 07:39:00 Writing pod manifest
2015/11/07 07:39:00 Setting up stage1
2015/11/07 07:39:00 Writing image manifest
2015/11/07 07:39:00 Wrote filesystem to /var/lib/rkt/pods/run/532d18a2-5fad-482a-b464-d87f47cc0099
2015/11/07 07:39:00 Writing image manifest
2015/11/07 07:39:00 Pivoting to filesystem /var/lib/rkt/pods/run/532d18a2-5fad-482a-b464-d87f47cc0099
2015/11/07 07:39:00 Execing /init
/ #
```

The following output shows you that the busybox container in `core-01` got the IP, `10.1.74.4`, which is in the `10.1.74.1/24` range allocated for `core-01`:

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 2E:FC:11:5C:D3:46
          inet addr:10.1.74.4  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::2cfc:11ff:fe5c:d346/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1472  Metric:1
          RX packets:12 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:1 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:1972 (1.9 KiB)  TX bytes:558 (558.0 B)
```

The following output shows you that the busybox container in `core-03` got the IP, `10.1.3.2`, which is in the `10.1.3.1/24` range allocated for `core-03`:

```
/ # ifconfig
eth0      Link encap:Ethernet  HWaddr 5E:84:DC:39:B4:CD
          inet addr:10.1.3.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::5c84:dcff:fe39:b4cd/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1472  Metric:1
          RX packets:30 errors:0 dropped:0 overruns:0 frame:0
          TX packets:7 errors:0 dropped:1 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:4185 (4.0 KiB)  TX bytes:558 (558.0 B)
```

The following output shows you a successful ping from container 1 on `core-01` to container 2 on `core-03`:

```
/ # ping -c1 10.1.3.2
PING 10.1.3.2 (10.1.3.2): 56 data bytes
64 bytes from 10.1.3.2: seq=0 ttl=60 time=0.786 ms

--- 10.1.3.2 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.786/0.786/0.786 ms
```

Summary

In this chapter, we covered different Container standards for Container runtime, networking, and orchestration. Having these standards is important from the industry perspective for interoperability reasons. Container runtime systems like Docker and Rkt were covered in detail. For Docker, the focus was on advanced concepts, and for Rkt, we covered the basics as Rkt is still in the early stages. Even though CoreOS is actively developing Rkt, CoreOS is committed to supporting Docker in its OS. It will be interesting to see how Docker and Rkt run together in CoreOS and how customers adopt the two Container runtime technologies. In the next chapter, we will cover Container orchestration.

References

- Notary GitHub: `https://github.com/docker/notary`
- Docker registry: `https://github.com/docker/distribution`
- Docker content trust documentation: `https://docs.docker.com/security/trust/content_trust/`
- Docker content trust blog: `https://blog.docker.com/2015/08/content-trust-docker-1-8/`
- The Update framework: `http://theupdateframework.com/`
- Cloud native compute foundation: `https://cncf.io`
- Open container initiative: `https://github.com/opencontainers`
- APPC specification: `https://github.com/appc`
- Libnetwork: `https://github.com/docker/libnetwork`
- Docker2aci: `https://github.com/appc/docker2aci`
- CoreOS Rkt documentation: `https://coreos.com/rkt/docs/latest/`
- Acbuild: `https://github.com/appc/acbuild`

Further reading and tutorials

- The CNI presentation: `https://www.youtube.com/watch?v=_-9kItVUUCw`
- Docker registry presentations: `https://www.youtube.com/watch?v=RnO9JnEO8tY` and `https://www.youtube.com/watch?v=cVsUhoJFPvQ`
- The Docker Notary presentation: `https://www.youtube.com/watch?v=JvjdfQC8jxM`
- The Container standards presentation: `http://containersummit.io/events/sf-2015/videos/container-ecosystem-standards-needs-and-progress`
- The Rkt and APPC presentation: `https://www.youtube.com/watch?v=C8Qpdrpm16Y`

8
Container Orchestration

As Containers became the basis of modern application development and deployment, it is necessary to deploy hundreds or thousands of Containers to a single data center cluster or data center clusters. The cluster could be an on-premises cluster or in a cloud. It is necessary to have a good Container orchestration system to deploy and manage Containers at scale.

The following topics will be covered in this chapter:

- The basics of modern application deployment
- Container orchestration with Kubernetes, Docker Swarm, and Mesos and their core concepts, installation, and deployment
- Comparison of popular orchestration solutions
- Application definition with Docker Compose
- Packaged Container Orchestration solutions—the AWS container service, Google container engine, and CoreOS Tectonic

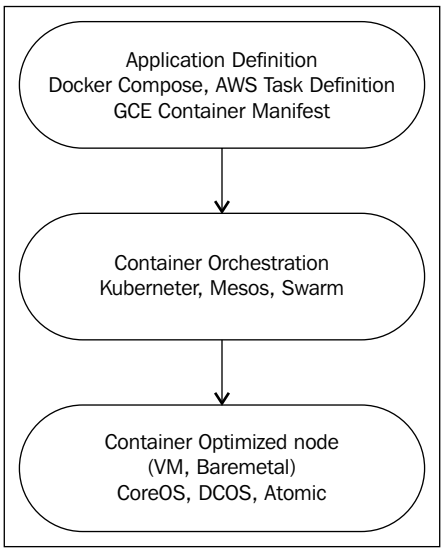
Modern application deployment

We covered the basics of Microservices in *Chapter 1*, *CoreOS Overview*. In cloud-based application development, infrastructure is treated as cattle rather than pet (`http://www.slideshare.net/randybias/pets-vs-cattle-the-elastic-cloud-story`). What this means is that the infrastructure is commonly a commodity hardware that can easily go bad and high availability needs to be handled at either the application layer or application Orchestration layer. High availability can be taken care of by having a combination of the load balancer and Orchestration system that monitors the health of services taking necessary actions such as respawning the service if it dies. Containers have the nice property of isolation and packaging that allows independent teams to develop individual components as Containers.

Companies can adopt a pay-as-you-grow model where they can scale their Containers as they grow. It is necessary to manage hundreds or thousands of Containers at scale. To do this efficiently, we need a Container Orchestration system. The following are some characteristics of a Container Orchestration system:

- It treats disparate infrastructure hardware as a collection and represents it as one single resource to the application
- It schedules Containers based on user constraints and uses the infrastructure in the most efficient manner
- It scales out containers dynamically
- It maintains high availability of services

There is a close relation between the application definition and Container Orchestration. The application definition is typically a manifest file describing the Containers that are part of the application and the services that the Container exposes. Container Orchestration is done based on the application definition. The Container Orchestrator operates on resources that could be a VM or bare metal. Typically, the nodes where Containers run are installed with Container-optimized OSes, such as CoreOS, DCOS, and Atomic. The following image shows you the relationship between the application definition, Container Orchestration, and Container-optimized nodes along with some examples of solutions in each category:



Container Orchestration

A basic requirement of Container orchestration is to efficiently deploy *M* containers into *N* compute resources.

The following are some problems that a Container Orchestration system should solve:

- It should schedule containers efficiently, giving enough control to the user to tweak scheduling parameters based on their need
- It should provide Container networking across the cluster
- Services should be able to discover each other dynamically
- Orchestration system should be able to handle service failures

We will cover Kubernetes, Docker Swarm, and Mesos in the following sections. Fleet is used internally by CoreOS for Container orchestration. Fleet has very minimal capabilities and works well for the deployment of critical system services in CoreOS. For very small deployments, Fleet can be used for Container orchestration, if necessary.

Kubernetes

Kubernetes is an open source platform for Container Orchestration. This was initially started by Google and now multiple vendors are working together in this open source project. Google has used Containers to develop and deploy applications in their internal data center and they had a system called Borg (`http://research.google.com/pubs/pub43438.html`) for cluster management. Kubernetes uses a lot of the concepts from Borg combined with modern technologies available now. Kubernetes is lightweight, works across almost all environments, and has a lot of industry traction currently.

Concepts of Kubernetes

Kubernetes has some unique concepts, and it will be good to understand them before diving deep into the architecture of Kubernetes.

Pods

Pods are a set of Containers that are scheduled together in a single node and need to work closely with each other. All containers in a Pod share the IPC namespace, network namespace, UTS namespace, and PID namespace. By sharing the IPC namespace, Containers can use IPC mechanisms to talk to each other.

By sharing the network namespace, Containers can use sockets to talk to each other, and all Containers in a Pod share a single IP address. By sharing the UTS namespace, volumes can be mounted to a Pod and all Containers can see these volumes. The following are some common application deployment patterns with Pods:

- **Sidecar pattern**: An example is an application container and logging container or application synchronizer container such as a Git synchronizer.
- **Ambassador pattern**: In this pattern, the application container and proxy container work together. When the application container changes, external services can still talk to the proxy container as before. An example is a redis application container with the redis proxy.
- **Adapter pattern**: In this pattern, there is an application container and adapter container that adapts to different environments. An example is a logging container that works as an adapter and changes with different cloud providers but the interface to the adapter container remains the same.

The smallest unit in Kubernetes is a Pod and Kubernetes takes care of scheduling the Pods.

The following is a Pod definition example with the NGINX Container and Git helper container:

```
apiVersion: v1
kind: Pod
metadata:
  name: www
spec:
  containers:
  - name: nginx
    image: nginx
  - name: git-monitor
    image: kubernetes/git-monitor
    env:
    - name: GIT_REPO
      value: http://github.com/some/repo.git
```

Networking

Kubernetes has the one IP per Pod approach. This approach was taken to avoid the pains associated with NAT to access Container services when Containers shared the host IP address. All Containers in a pod share the same IP address. Pods across nodes can talk to each other using different techniques such as cloud-based routing from cloud providers, Flannel, Weave, Calico, and others. The end goal is to have Networking as a plugin within Kubernetes and the user can choose the plugin based on their needs.

Services

Services are an abstraction that Kubernetes provides to logically combine Pods that provide similar functionality. Typically, Labels are used as selectors to create services from Pods. As Pods are ephemeral, Kubernetes creates a service object with its own IP address that always remains permanent. Kubernetes takes care of load balancing for multiple pods.

The following is an example service:

```
{
    "kind": "Service",
    "apiVersion": "v1",
    "metadata": {
        "name": "my-service"
    },
    "spec": {
        "selector": {
            "app": "MyApp"
        },
        "ports": [
            {
                "protocol": "TCP",
                "port": 80,
                "targetPort": 9376
            }
        ]
    }
}
```

In the preceding example, we created a `my-service` service that groups all pods with a `Myapp` label. Any request to the `my-service` service's IP address and port number `80` will be load balanced to all the pods with the `Myapp` label and redirected to port number `9376`.

Services need to be discovered internally or externally based on the type of service. An example of internal discovery is a web service needing to talk to a database service. An example of external discovery is a web service that gets exposed to the outside world.

For internal service discovery, Kubernetes provides two options:

- **Environment variable**: When a new Pod is created, environment variables from older services can be imported. This allows services to talk to each other. This approach enforces ordering in service creation.

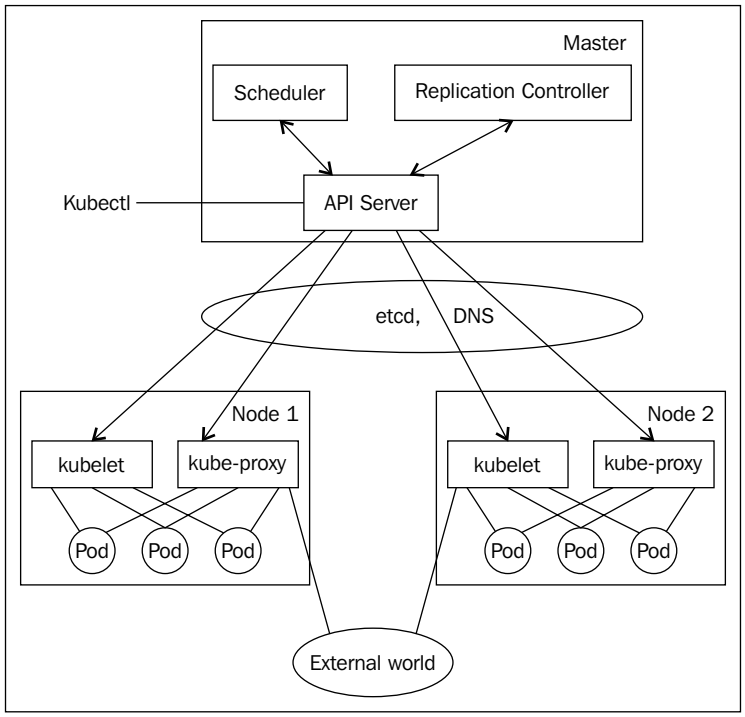
- **DNS**: Every service registers to the DNS service; using this, new services can find and talk to other services. Kubernetes provides the `kube-dns` service for this.

For external service discovery, Kubernetes provides two options:

- **NodePort**: In this method, Kubernetes exposes the service through special ports (30000-32767) of the node IP address.
- **Loadbalancer**: In this method, Kubernetes interacts with the cloud provider to create a load balancer that redirects the traffic to the Pods. This approach is currently available with GCE.

Kubernetes architecture

The following diagram shows you the different software components of the Kubernetes architecture and how they interact with each other:



The following are some notes on the Kubernetes architecture:

- The master node hosts the Kubernetes control services. Slave nodes run the pods and are managed by master nodes. There can be multiple master nodes for redundancy purposes and to scale master services.
- Master nodes run the critical services such as the Scheduler, Replication controller, and API server. Slave nodes run the critical services such as Kubelet and Kube-proxy.
- User interaction with Kubernetes is through Kubectl, which uses standard Kubernetes-exposed APIs.
- The Kubernetes scheduler takes care of scheduling the pods in the nodes based on the constraints specified in the Pod manifest.
- The replication controller is necessary to maintain high availability of Pods and create multiple instances of pods as specified in the replication controller manifest.
- The API server in the master node talks to Kubelet of each slave node to provision the pods.
- Kube-proxy takes care of service redirection and load balancing the traffic to the Pods.
- Etcd is used as a shared data repository for all nodes to communicate with each other.
- DNS is used for service discovery.

Kubernetes installation

Kubernetes can be installed on baremetal, VM, or in cloud providers such as AWS, GCE, and Azure. We can decide on the choice of the host OS on any of these systems. In this chapter, all the examples will use CoreOS as the host OS. As Kubernetes consists of multiple components such as the API server, scheduler, replication controller, kubectl, and kubeproxy spread between master and slave nodes, the manual installation of the individual components would be complicated. There are scripts provided by Kubernetes and its users that automate some of the node setup and software installation. The latest stable version of Kubernetes as of October 2015 is 1.0.7 and all examples in this chapter are based on the 1.0.7 version.

Non-Coreos Kubernetes installation

For non-Coreos-based Kubernetes installation, the procedure is straightforward:

1. Find the Kubernetes release necessary from `https://github.com/kubernetes/kubernetes/releases`.
2. Download `kubernetes.tar.gz` for the appropriate version and unzip them.
3. Set `KUBERNETES_PROVIDER` as one of these (AWS, GCE, Vagrant, and so on)
4. Change the cluster size and any other configuration parameter in the `cluster` directory.
5. Run `cluster/kube-up.sh`.

Kubectl installation

Kubectl is the CLI client to interact with Kubernetes. Kubectl is not installed by default. Kubectl can be installed in either the client machine or the kubernetes master node.

The following command can be used to install kubectl. It is needed to match kubectl version with Kubernetes version:

```
ARCH=linux; wget https://storage.googleapis.com/kubernetes-release/
release/v1.0.7/bin/$ARCH/amd64/kubectl
```

If Kubectl is installed in the client machine, we can use the following command to proxy the request to the Kubernetes master node:

```
ssh -f -nNT -L 8080:127.0.0.1:8080 core@<control-external-ip>
```

Vagrant installation

I used the procedure at `https://github.com/pires/kubernetes-vagrant-coreos-cluster` to create a Kubernetes cluster running on the Vagrant environment with CoreOS. I initially tried this in Windows. As I faced the issue mentioned in `https://github.com/pires/kubernetes-vagrant-coreos-cluster/issues/158`, I moved to the Vagrant environment running on Ubuntu Linux.

The following are the commands:

```
Git clone https://github.com/pires/kubernetes-vagrant-coreos-cluster.git
Cd coreos-container-platform-as-a-service/vagrant
Vagrant up
```

The following output shows the two running Kubernetes nodes:

```
sreeni@ubuntu:~/coreos-container-platform-as-a-service/vagrant$ kubectl get nodes
NAME           LABELS                                    STATUS
172.17.8.102   kubernetes.io/hostname=172.17.8.102   Ready
172.17.8.103   kubernetes.io/hostname=172.17.8.103   Ready
```

GCE installation

I used the procedure at `https://github.com/rimusz/coreos-multi-node-k8s-gce` to create a Kubernetes cluster running in GCE with CoreOS.

The following are the commands:

```
git clone https://github.com/rimusz/coreos-multi-node-k8s-gce
cd coreos-multi-node-k8s-gce
```

In the `settings` file, change `project`, `zone`, `node count`, and any other necessary changes.

Run the following three scripts in the same order:

```
1-bootstrap_cluster.sh
2-get_k8s_fleet_etcd.sh
3-install_k8s_fleet_units.sh
```

The following output shows the cluster composed of three nodes:

```
smakam14@jungle1:~$ kubectl get nodes
NAME                                          LABELS                                                        STATUS    AGE
k8s-node-1.c.stunning-chain-108807.internal   kubernetes.io/hostname=k8s-node-1.c.stunning-chain-108807.internal   Ready     23h
k8s-node-2.c.stunning-chain-108807.internal   kubernetes.io/hostname=k8s-node-2.c.stunning-chain-108807.internal   Ready     23h
k8s-node-3.c.stunning-chain-108807.internal   kubernetes.io/hostname=k8s-node-3.c.stunning-chain-108807.internal   Ready     23h
```

The following output shows the Kubernetes client and server versions:

```
smakam14@jungle1:~$ kubectl version
Client Version: version.Info{Major:"1", Minor:"1", GitVersion:"v1.1.1", GitCommit:"92635e23dfafb2ddc828c8ac6c03c7a7205a84d8", GitTreeSta
te:"clean"}
Server Version: version.Info{Major:"1", Minor:"1", GitVersion:"v1.1.1", GitCommit:"92635e23dfafb2ddc828c8ac6c03c7a7205a84d8", GitTreeSta
te:"clean"}
```

The following output shows the Kubernetes services running in master and slave nodes:

```
core@k8s-master ~ $ fleetctl list-units
UNIT                            MACHINE                 ACTIVE  SUB
kube-apiserver.service          a0902c39.../10.240.0.2  active  running
kube-controller-manager.service a0902c39.../10.240.0.2  active  running
kube-kubelet.service            1dcef688.../10.240.0.5  active  running
kube-kubelet.service            dee9c694.../10.240.0.3  active  running
kube-kubelet.service            ef45a03b.../10.240.0.4  active  running
kube-proxy.service              1dcef688.../10.240.0.5  active  running
kube-proxy.service              dee9c694.../10.240.0.3  active  running
kube-proxy.service              ef45a03b.../10.240.0.4  active  running
kube-scheduler.service          a0902c39.../10.240.0.2  active  running
```

The script used in this example uses Fleet to orchestrate Kubernetes services. As we can see in the preceding image, the API server, controller, and scheduler run in the master node and kubelet and proxy run in the slave nodes. There are three copies of kubelet and kube-proxy, one each for every slave node.

AWS installation

I used the procedure at `https://coreos.com/kubernetes/docs/latest/kubernetes-on-aws.html` to create the Kubernetes CoreOS cluster running on AWS.

The first step is to install the kube-aws tool:

```
Git clone https://github.com/coreos/coreos-kubernetes/releases/download/v0.1.0/kube-aws-linux-amd64.tar.gz
```

Unzip and copy kube-aws to an executable path. Make sure that `~/.aws/credentials` is updated with your credentials.

Create a default `cluster.yaml` file:

```
curl --silent --location https://raw.githubusercontent.com/coreos/coreos-kubernetes/master/multi-node/aws/cluster.yaml.example > cluster.yaml
```

Modify `cluster.yaml` with your `keyname`, `region`, and `externaldnsname`; `externaldnsname` matters for external access only.

To deploy the cluster, we can perform the following:

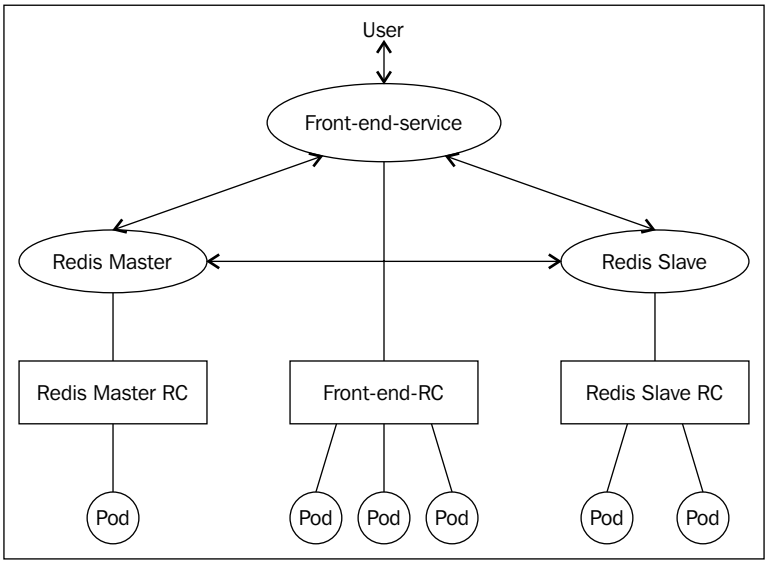
```
Kube-aws up
```

The following output shows the two nodes that are part of the Kubernetes cluster:

```
smakam14@jungle1:~/aws1$ kubectl get nodes
NAME                                        LABELS                                                      STATUS    AGE
ip-10-0-0-101.us-west-2.compute.internal    kubernetes.io/hostname=ip-10-0-0-101.us-west-2.compute.internal    Ready     9h
ip-10-0-0-102.us-west-2.compute.internal    kubernetes.io/hostname=ip-10-0-0-102.us-west-2.compute.internal    Ready     9h
```

An example of a Kubernetes application

The following diagram illustrates the guestbook example that we will use to illustrate the different Kubernetes concepts discussed in the previous sections. This example is based on the reference at `http://kubernetes.io/v1.1/examples/guestbook/README.html`:



The following are some notes on this guestbook application:

- This application uses the php frontend with the redis master and slave backend to store the guestbook database
- Frontend RC creates three instances of the `kubernetes/example-guestbook-php-redis` container
- Redis-master RC creates one instance of the redis container
- Redis-slave RC creates two instances of the `kubernetes/redis-slave` container

For this example, I used the cluster created in the previous section with Kubernetes running on AWS with CoreOS. There is one master node and two slave nodes.

Let's look at the nodes:

```
smakam14@jungle1:~/kubernetes107/kubernetes/examples/guestbook$ kubectl get nodes
NAME                                        LABELS                                                    STATUS    AGE
ip-10-0-0-101.us-west-2.compute.internal    kubernetes.io/hostname=ip-10-0-0-101.us-west-2.compute.internal   Ready     1d
ip-10-0-0-102.us-west-2.compute.internal    kubernetes.io/hostname=ip-10-0-0-102.us-west-2.compute.internal   Ready     1d
```

In this example, the Kubernetes cluster uses flannel to communicate across pods. The following output shows the flannel subnet allocated to each node in the cluster:

```
core@ip-10-0-0-50 ~ $ etcdctl ls / --recursive | grep subnet
/coreos.com/network/subnets
/coreos.com/network/subnets/10.2.10.0-24
/coreos.com/network/subnets/10.2.26.0-24
/coreos.com/network/subnets/10.2.55.0-24
```

The following are the commands necessary to start the application:

```
kubectl create -f redis-master-controller.yaml
kubectl create --validate=false -f redis-master-service.yaml
kubectl create -f redis-slave-controller.yaml
kubectl create --validate=false -f redis-slave-service.yaml
kubectl create -f frontend-controller.yaml
kubectl create --validate=false -f frontend-service.yaml
```

Let's look at the list of pods:

```
smakam14@jungle1:~/kubernetes107/kubernetes/examples/guestbook$ kubectl get pods
NAME                READY     STATUS    RESTARTS   AGE
frontend-8mwjm      1/1       Running   0          13m
frontend-8riby      1/1       Running   0          13m
frontend-bko06      1/1       Running   0          13m
redis-master-dxjqi  1/1       Running   0          14m
redis-slave-2rvtm   1/1       Running   0          14m
redis-slave-ry90e   1/1       Running   0          14m
```

The preceding output shows three instances of the php frontend, one instance of the redis master, and two instances of the redis slave.

Let's look at the list of RC:

```
smakam14@jungle1:~/kubernetes107/kubernetes/examples/guestbook$ kubectl get rc
CONTROLLER     CONTAINER(S)   IMAGE(S)                                  SELECTOR            REPLICAS   AGE
frontend       php-redis      kubernetes/example-guestbook-php-redis:v2   name=frontend       3          18m
redis-master   master         redis                                     name=redis-master   1          18m
redis-slave    worker         kubernetes/redis-slave:v2                 name=redis-slave    2          18m
```

The preceding output shows the replication count per pod. Frontend has three replicas, `redis-master` has one replica, and `redis-slave` has two replicas, as we requested.

Let's look at the list of services:

```
smakam14@jungle1:~/kubernetes107/kubernetes/examples/guestbook$ kubectl get services
NAME           CLUSTER_IP    EXTERNAL_IP   PORT(S)    SELECTOR            AGE
frontend       10.3.0.221    nodes         80/TCP     name=frontend       17m
kubernetes     10.3.0.1      <none>        443/TCP    <none>              1d
redis-master   10.3.0.218    <none>        6379/TCP   name=redis-master   18m
redis-slave    10.3.0.141    <none>        6379/TCP   name=redis-slave    18m
```

In the preceding output, we can see the three services comprising the guestbook application.

For internal service discovery, this example uses `kube-dns`. The following output shows the `kube-dns` RC running:

```
smakam14@jungle1:~/aws1$ kubectl --namespace=kube-system get rc
CONTROLLER    CONTAINER(S)   IMAGE(S)                                          SELECTOR                        REPLICAS   AGE
kube-dns-v9   etcd           gcr.io/google_containers/etcd:2.0.9               k8s-app=kube-dns,version=v9     1          1d
              kube2sky       gcr.io/google_containers/kube2sky:1.11
              skydns         gcr.io/google_containers/skydns:2015-03-11-001
              healthz        gcr.io/google_containers/exechealthz:1.0
```

For external service discovery, I modified the example to use the `NodePort` mechanism, where one of the internal ports gets exposed. The following is the new `frontend-service.yaml` file:

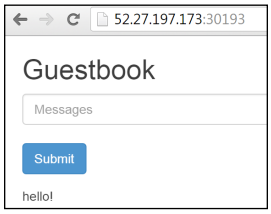
```
apiVersion: v1
kind: Service
metadata:
  name: frontend
  labels:
    name: frontend
spec:
  # if your cluster supports it, uncomment the following to
automatically create
  # an external load-balanced IP for the frontend service.
  type: NodePort
```

```
  ports:
    # the port that this service should serve on
  - port: 80
  selector:
    name: frontend
```

The following is the output when we start the frontend service with the `NodePort` type. The output shows that the service is exposed using port `30193`:

```
smakam14@jungle1:~/kubernetes107/kubernetes/examples/guestbook$ kubectl create --validate=false -f f
rontend-service1.yaml
You have exposed your service on an external port on all nodes in your
cluster.  If you want to expose this service to the external internet, you may
need to set up firewall rules for the service port(s) (tcp:30193) to serve traffic.
```

Once we expose port `30193` using the AWS firewall, we can access the guestbook application as follows:



Let's look at the application containers in `Node1`:

```
core@ip-10-0-0-101 ~ $ docker ps | grep guestbook
14a76c5aa9ec        kubernetes/example-guestbook-php-redis:v2        "/bin/sh -c /run.sh"     32 minutes ago      Up 31 minutes
            k8s_php-redis.81fd265b_frontend-8mwjm_default_b2212ae7-887a-11e5-969e-02dd3d0731eb_4a4ffd69
core@ip-10-0-0-101 ~ $ docker ps | grep redis
14a76c5aa9ec        kubernetes/example-guestbook-php-redis:v2        "/bin/sh -c /run.sh"     32 minutes ago      Up 32 minutes
            k8s_php-redis.81fd265b_frontend-8mwjm_default_b2212ae7-887a-11e5-969e-02dd3d0731eb_4a4ffd69
```

Let's look at the application containers in `Node2`:

```
core@ip-10-0-0-102 ~ $ docker ps  | grep guest
750ea46cb141        kubernetes/example-guestbook-php-redis:v2   "/bin/sh -c /run.sh"     35 minutes ago      Up 35 minutes
          k8s_php-redis.81fd265b_frontend-8riby_default_b224cecd-887a-11e5-969e-02dd3d0731eb_5626cb55
b01a7cd58fc7        kubernetes/example-guestbook-php-redis:v2   "/bin/sh -c /run.sh"     35 minutes ago      Up 35 minutes
          k8s_php-redis.81fd265b_frontend-bko06_default_b2211265-887a-11e5-969e-02dd3d0731eb_ea36f77b
core@ip-10-0-0-102 ~ $ docker ps  | grep kubernetes.redis
7175506b84a5        kubernetes/redis-slave:v2                   "/bin/sh -c /run.sh"     35 minutes ago      Up 35 minutes
          k8s_worker.c5431f94_redis-slave-2rvtm_default_b0beac98-887a-11e5-969e-02dd3d0731eb_0c7ef947
80f170819849        kubernetes/redis-slave:v2                   "/bin/sh -c /run.sh"     35 minutes ago      Up 35 minutes
          k8s_worker.c5431f94_redis-slave-ry90e_default_b0bde51b-887a-11e5-969e-02dd3d0731eb_b51486a9
```

The preceding output accounts for three instances of frontend, one instance of redis master, and two instances of redis slave.

To illustrate how the replication controller maintains the pod replication count, I went and stopped the guestbook frontend Docker Container in one of the nodes, as shown in the following image:

```
core@ip-10-0-0-101 ~ $ docker stop 14a76c5aa9ec
14a76c5aa9ec
```

Kubernetes RC detects that the Pod is not running and restarts the Pod. This can be seen in the restart count for one of the guestbook pods, as shown in the following image:

```
smakam14@jungle1:~/kubernetes107/kubernetes/examples/guestbook$ kubectl get pods
NAME                READY     STATUS    RESTARTS   AGE
frontend-8mwjm      1/1       Running   1          38m
frontend-8riby      1/1       Running   0          38m
frontend-bko06      1/1       Running   0          38m
```

To do some basic debugging, we can log in to the pods or containers themselves. The following example shows you how we can get inside the Pod:

```
smakam14@jungle1:~/kubernetes107/kubernetes/examples/guestbook$ kubectl exec frontend-8mwjm -i -t -- bash -il
root@frontend-8mwjm:/# ifconfig
eth0      Link encap:Ethernet  HWaddr 02:42:0a:02:37:04
          inet addr:10.2.55.4  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:aff:fe02:3704/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:8973  Metric:1
          RX packets:33 errors:0 dropped:0 overruns:0 frame:0
          TX packets:30 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:3234 (3.2 KB)  TX bytes:2547 (2.5 KB)
```

The preceding output shows the IP address in the guestbook pod, which agrees with the flannel subnet allocated to that node, as shown in the Flannel output in the beginning of this example.

Another command that's useful for the debugging is `kubectl logs` as follows:

```
smakam14@jungle1:~/kubernetes107/kubernetes/examples/guestbook$ kubectl logs frontend-8mwjm
2015-11-11 14:25:09,026 CRIT Supervisor running as root (no user in config file)
2015-11-11 14:25:09,026 WARN Included extra file "/etc/supervisor/conf.d/supervisord-apache2.conf" during parsing
```

Kubernetes with Rkt

By default, Kubernetes works with Container runtime Docker. The architecture of Kubernetes allows other Container runtime such as Rkt to work with Kubernetes. There is active work going on (`https://github.com/kubernetes/kubernetes/tree/master/docs/getting-started-guides/rkt`) to integrate Kubernetes with Rkt and CoreOS.

Kubernetes 1.1 update

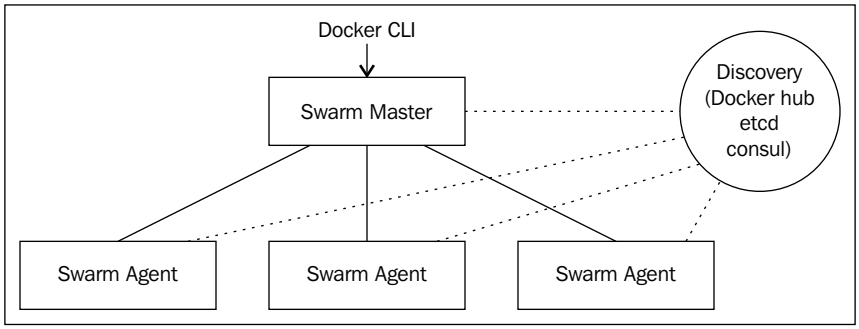
Kubernetes released 1.1 version (`http://blog.kubernetes.io/2015/11/Kubernetes-1-1-Performance-upgrades-improved-tooling-and-a-growing-community.html`) in November 2015. Significant additions in 1.1 are increased performance, auto-scaling, and job objects for the batching tasks.

Docker Swarm

Swarm is Docker's native Orchestration solution. The following are some properties of Docker Swarm:

- Rather than managing individual Docker nodes, the cluster can be managed as a single entity.
- Swarm has a built-in scheduler that decides the placement of Containers in the cluster. Swarm uses user-specific constraints and affinities (`https://docs.docker.com/swarm/scheduler/filter/`) to decide the Container placement. Constraints could be CPU and memory, and affinity are parameters to group related Containers together. Swarm also has the provision to take its scheduler out and work with other schedulers such as Kubernetes.

The following image shows the Docker Swarm architecture:



The following are some notes on the Docker Swarm architecture:

- The **Swarm Master** takes care of scheduling Docker Containers based on the scheduling algorithm, constraints, and affinities. Supported algorithms are spread, `binpack`, and `random`. The default algorithm is spread. Multiple Swarm masters can be run in parallel to provide high availability. The Spread scheduling is used to distribute workloads evenly. The binpack scheduling is used to utilize each node fully before scheduling on another node.

- The **Swarm Agent** runs in each node and communicates to the **Swarm Master**.
- There are different approaches available for Swarm worker nodes to discover the **Swarm Master**. Discovery is necessary because the **Swarm Master** and agents run on different nodes and Swarm agents are not started by the **Swarm Master**. It is necessary for Swarm agents and the **Swarm Master** to discover each other in order to understand that they are part of the same cluster. Available discovery mechanisms are Docker hub, Etcd, Consul, and others.
- Docker Swarm integrates with the Docker machine to ease the creation of Docker nodes. Docker Swarm integrates with Docker compose for multicontainer application orchestration.
- With the Docker 1.9 release, Docker Swarm integrates with multi-host Docker networking that allows Containers scheduled across hosts to talk to each other.

The Docker Swarm installation

A prerequisite for this example is to install Docker 1.8.1 and Docker-machine 0.5.0. I used the procedure at `https://docs.docker.com/swarm/install-w-machine/` to create a single Docker Swarm master with two Docker Swarm agent nodes. The following are the steps:

1. Create a discovery token.
2. Create a Swarm master node with the created discovery token.
3. Create two Swarm agent nodes with the created discovery token.

By setting the environment variable to `swarm-master`, as shown in the following command we can control the Docker swarm cluster using regular Docker commands:

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

Let's look at the `docker info` output on the Swarm cluster:

```
Containers: 4
Images: 5
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 3
 swarm-agent-00: 192.168.99.103:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.1.12-boot2docker, operatingsystem=Boot2Docker 1.9.0 (TCL 6.4); master : 16e4a2a
 - Tue Nov  3 19:49:22 UTC 2015, provider=virtualbox, storagedriver=aufs
 swarm-agent-01: 192.168.99.104:2376
  └ Containers: 1
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.1.12-boot2docker, operatingsystem=Boot2Docker 1.9.0 (TCL 6.4); master : 16e4a2a
 - Tue Nov  3 19:49:22 UTC 2015, provider=virtualbox, storagedriver=aufs
 swarm-master: 192.168.99.102:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.1.12-boot2docker, operatingsystem=Boot2Docker 1.9.0 (TCL 6.4); master : 16e4a2a
 - Tue Nov  3 19:49:22 UTC 2015, provider=virtualbox, storagedriver=aufs
CPUs: 3
Total Memory: 3.064 GiB
Name: 27a599a19a65
```

The preceding output shows that three nodes (one master and two agents) are in the cluster and that four containers are running in the cluster. The `swarm-master` node has two containers and the `swarm-agent` node has one container each. These containers are used to manage the Swarm service. Application containers are scheduled only in Swarm agent nodes.

Let's look at the individual containers in the master node. This shows the master and agent services running:

```
docker@swarm-master:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              P
ORTS                                NAMES
7f521c987058        swarm:latest        "/swarm join --advert"   31 seconds ago      Up 30 seconds       2
375/tcp                             swarm-agent
27a599a19a65        swarm:latest        "/swarm manage --tlsv"   31 seconds ago      Up 31 seconds       2
375/tcp, 0.0.0.0:3376->3376/tcp     swarm-agent-master
```

Let's look at the container running in the agent node. This shows the `swarm agent` running:

```
docker@swarm-agent-00:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
965043240c96        swarm:latest        "/swarm join --advert"   17 seconds ago      Up 17 seconds       2375/tcp            swarm-agen
t
```

An example of Docker Swarm

To illustrate the Docker Swarm Container orchestration, let's start four `nginx` containers:

```
docker run -d --name nginx1 nginx
docker run -d --name nginx2 nginx
docker run -d --name nginx3 nginx
docker run -d --name nginx4 nginx
```

From the following output, we can see that the four containers are spread equally between `swarm-agent-00` and `swarm-agent-01`. The default `spread` scheduling strategy has been used here:

```
sreeni@ubuntu:~$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
b68319c9d50f        nginx               "nginx -g 'daemon off"   39 minutes ago      Up 39 minutes       80/tcp, 443/tcp     swarm-agen
t-00/nginx4
0ff325bd1906        nginx               "nginx -g 'daemon off"   39 minutes ago      Up 39 minutes       80/tcp, 443/tcp     swarm-agen
t-01/nginx3
e7de3bbe6a92        nginx               "nginx -g 'daemon off"   39 minutes ago      Up 39 minutes       80/tcp, 443/tcp     swarm-agen
t-01/nginx2
9285bcaf9344        nginx               "nginx -g 'daemon off"   40 minutes ago      Up 40 minutes       80/tcp, 443/tcp     swarm-agen
t-00/nginx1
```

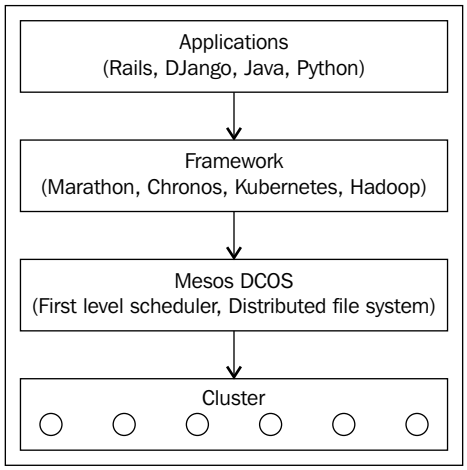
The following output shows the overall Container count across the cluster that includes the master and two agent nodes. The container count eight includes Swarm service containers as well as nginx application containers:

```
sreeni@ubuntu:~$ docker info
Containers: 8
Images: 6
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 3
 swarm-agent-00: 192.168.99.103:2376
  └ Containers: 3
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.1.12-boot2docker, operatingsystem=Boot2Docker 1.9.0 (TCL 6.4); master : 16e4a2a
 - Tue Nov  3 19:49:22 UTC 2015, provider=virtualbox, storagedriver=aufs
 swarm-agent-01: 192.168.99.104:2376
  └ Containers: 3
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.1.12-boot2docker, operatingsystem=Boot2Docker 1.9.0 (TCL 6.4); master : 16e4a2a
 - Tue Nov  3 19:49:22 UTC 2015, provider=virtualbox, storagedriver=aufs
 swarm-master: 192.168.99.102:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.1.12-boot2docker, operatingsystem=Boot2Docker 1.9.0 (TCL 6.4); master : 16e4a2a
 - Tue Nov  3 19:49:22 UTC 2015, provider=virtualbox, storagedriver=aufs
CPUs: 3
Total Memory: 3.064 GiB
Name: 27a599a19a65
```

Mesos

Apache Mesos is an open source clustering software. Mesosphere's DCOS is the commercial version of Apache Mesos. Mesos combines the Clustering OS and Cluster manager. Clustering OS is responsible for representing resources from multiple disparate computers in one single resource over which applications can be scheduled. The cluster manager is responsible for scheduling the jobs in the cluster. The same cluster can be used for different workloads such as Hadoop and Spark. There is a two-level scheduling within Mesos. The first-level scheduling does resource allocation among frameworks and the framework takes care of scheduling the jobs within that particular framework. Each framework is an application category such as Hadoop, Spark, and others. For general purpose applications, the best framework available is Marathon. Marathon is a distributed INIT and HA system to schedule containers. The Chronos framework is like a Cron job and this framework is suitable to run shorter workloads that need to be run periodically. The Aurora framework provides you with a much more fine-grained control for complex jobs.

The following image shows you the different layers in the Mesos architecture:



Comparing Kubernetes, Docker Swarm, and Mesos

Even though all these solutions (Kubernetes, Docker Swarm, and Mesos) do application Orchestration, there are many differences in their approach and use cases. I have tried to summarize the differences based on their latest available release. All these Orchestration solutions are under active development, so the feature set can vary going forward. This table is updated as of October 2015:

Feature	Kubernetes	Docker Swarm	Mesos
Deployment unit	Pods	Container	Process or Container
Container runtime	Docker and Rkt.	Docker.	Docker; there is some discussion ongoing on Mesos with Rkt.
Networking	Each container has an IP address, and can use external network plugins.	Initially, this did Port forwarding with a common agent IP address. With Docker 1.9, it uses Overlay networking and per container IP address. It can use external network plugins.	Initially, this did Port forwarding with a common agent IP address. Currently, it works on per Container IP. integration with Calico.
Workloads	Homogenous workload. With namespaces, multiple virtual clusters can be created.	Homogenous workload.	Multiple frameworks such as Marathon, Aurora, and Hadoop can be run in parallel.
Service discovery	This can use either environment variable-based discovery or Kube-dns for dynamic discovery.	Static with modification of `/etc/hosts`. A DNS approach is planned in the future.	DNS-based approach to discover services dynamically.
High availability	With a replication controller, services are highly available. Service scaling can be done easily.	Service high availability is not yet implemented.	Frameworks take care of service high availability. For example, Marathon has the `Init.d` system to run containers.
Maturity	Relatively new. The first production release was done a few months before.	Relatively new. The first production release was done a few months before.	Pretty stable and used in big production environments.

Feature	Kubernetes	Docker Swarm	Mesos
Complexity	Easy.	Easy.	This is a little difficult to set up.
Use case	This is more suitable for homogenous workloads.	Presenting Docker frontend makes it attractive for Docker users not needing to learn any new management interface.	Suitable for heterogeneous workloads.

Kubernetes can be run as a framework on top of Mesos. In this case, Mesos does the first-level scheduling for Kubernetes and Kubernetes schedules and manages applications scheduled. This project (`https://github.com/mesosphere/kubernetes-mesos`) is dedicated to running Kubernetes on top of Mesos.

There is work ongoing to integrate Docker Swarm with Kubernetes so that Kubernetes can be used as a scheduler and process manager for the cluster while users can still use the Docker interface to manage containers using Docker Swarm.

Application definition

When an application is composed of multiple Containers, it is useful to represent each Container property along with its dependencies in a single JSON or YAML file so that the application can be instantiated as a whole rather than instantiating each Container of the application separately. The application definition file takes care of defining the multicontainer application. Docker-compose defines both the application file and runtime to instantiate containers based on the application file.

Docker-compose

Docker-compose provides you with an application definition format, and when we run the tool, Docker-compose takes care of parsing the application definition file and instantiating the Containers taking care of all the dependencies.

Docker-compose has the following advantages and use cases:

- It gives a simple approach to specify an application's manifest that contains multiple containers along with their constraints and affinities
- It integrates well with Dockerfile, Docker Swarm, and multihost networking
- The same compose file can be adapted to different environments using environment variables

A single-node application

The following example shows you how to build a multicontainer WordPress application with a WordPress and MySQL container.

The following is the `docker-compose.yml` file defining the Containers and their properties:

```
wordpress:
  image: wordpress
  ports:
   - "8080:80"
  environment:
    WORDPRESS_DB_HOST: "composeword_mysql_1:3306"
    WORDPRESS_DB_PASSWORD: mysql
mysql:
  image: mysql
  environment:
    MYSQL_ROOT_PASSWORD: mysql
```

The following command shows you how to start the application using `docker-compose`:

```
docker-compose -p composeword -f docker-compose.yml up -d
```

The following is the output of the preceding command:

```
smakam14@jungle1:~/composeword$ docker-compose -p composeword -f docker-compose.yml up -d
Creating composeword_wordpress_1...
Creating composeword_mysql_1... _
```

Containers are prefixed with a keyword specified in the `-p` option. In the preceding example, we have used `composeword_mysql_1` as the hostname, and the IP address is derived dynamically from the container using this and updated in `/etc/hosts`.

The following output shows the running containers of the wordpress application:

```
smakam14@jungle1:~/composeword$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS                    NAMES
f31d3e91f3e8        mysql               "/entrypoint.sh mysql"   5 seconds ago       Up 4 seconds        3306/tcp                 compos
eword_mysql_1
2f727dfeed66        wordpress           "/entrypoint.sh apach"   6 seconds ago       Up 5 seconds        0.0.0.0:8080->80/tcp     compos
eword_wordpress_1
```

The following output is the `/etc/hosts` output in the wordpress container; the one which shows that the IP address of the MySQL container is dynamically updated:

```
root@2f727dfeed66:/var/www/html# cat /etc/hosts | grep mysql
172.18.0.3      composeword_mysql_1
```

A multinode application

I used the example at `https://docs.docker.com/engine/userguide/networking/get-started-overlay/` to create a web application spanning multiple nodes using `docker-compose`. In this case, `docker-compose` is integrated with Docker Swarm and Docker multihost networking.

The prerequisite for this example is to have a working Docker Swarm cluster and Docker version 1.9+.

The following command creates the multihost counter application. This application has a web container as the frontend and a mongo container as the backend. These commands have to be executed against the Swarm cluster:

```
docker-compose -p counter -x-networking up -d
```

The following is the output of the preceding command:

```
sreeni@ubuntu:~/compose$ docker-compose -p counter --x-networking up -d
Creating network "counter" with driver "None"
Creating counter_web_1
Creating counter_mongo_1
```

The following output shows the overlay network `counter` created as part of this application:

```
docker@mhs-demo0:~$ docker network ls
NETWORK ID          NAME                DRIVER
0e5fca42f541        counter             overlay
b79a7294a131        my-net              overlay
b1097320a489        bridge              bridge
3f5cb7139b49        none                null
425bbde2136e        host                host
0f0cf336f062        docker_gwbridge     bridge
```

The following output shows the running Containers in the Swarm cluster:

```
sreeni@ubuntu:~/compose$ docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS
PORTS                     NAMES
8bd2185d4b29        mongo                          "/entrypoint.sh mongo"   29 minutes ago      Up 29 minutes
27017/tcp                 mhs-demo1/counter_mongo_1
5e8b80ba28b8        bfirsh/compose-mongodb-demo    "/bin/sh -c 'python a"   29 minutes ago      Up 29 minutes
192.168.99.101:80->5000/tcp   mhs-demo0/counter_web_1
```

The following output shows the Swarm cluster information. There are in total five containers—three of them are Swarm service containers and two of them are the preceding application containers:

```
sreeni@ubuntu:~/compose$ docker info
Containers: 5
Images: 6
Role: primary
Strategy: spread
Filters: health, port, dependency, affinity, constraint
Nodes: 2
 mhs-demo0: 192.168.99.101:2376
  └ Containers: 3
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.1.12-boot2docker, operatingsystem=Boot2Docker 1.9.0 (TCL 6.
4); master : 16e4a2a - Tue Nov  3 19:49:22 UTC 2015, provider=virtualbox, storagedriver=aufs
 mhs-demo1: 192.168.99.102:2376
  └ Containers: 2
  └ Reserved CPUs: 0 / 1
  └ Reserved Memory: 0 B / 1.021 GiB
  └ Labels: executiondriver=native-0.2, kernelversion=4.1.12-boot2docker, operatingsystem=Boot2Docker 1.9.0 (TCL 6.
4); master : 16e4a2a - Tue Nov  3 19:49:22 UTC 2015, provider=virtualbox, storagedriver=aufs
CPUs: 2
Total Memory: 2.043 GiB
```

The following output shows the working web application:

```
docker@mhs-demo0:~$ curl localhost
<h1>This page has been visited 2 times!</h1>
```

Packaged Container Orchestration solutions

There are many components necessary for the deployment of a distributed microservice application at scale. The following are some of the important components:

- An infrastructure cluster
- A Container-optimized OS
- A Container orchestrator with a built-in scheduler, service discovery, and networking
- Storage integration
- Multitenant capability with authentication
- An API at all layers to ease management

Cloud providers such as Amazon and Google already have an ecosystem to manage VMs, and their approach has been to integrate Containers and Container orchestration into their IaaS offering so that Containers play well with their other tools. The AWS Container service and Google Container engine fall in this category. The focus of CoreOS has been to develop a secure Container-optimized OS and open source tools for distributed application development. CoreOS realized that integrating their offering with Kubernetes would give their customers an integrated solution, and Tectonic provides this integrated solution.

There are a few other projects such as OpenStack Magnum (`https://github.com/openstack/magnum`) and Cisco's Mantl (`https://mantl.io/`) that falls under this category of managed Container Orchestration. We have not covered these in this chapter.

The AWS Container service

The AWS **EC2 Container Service** (**ECS**) is a Container Orchestration service from AWS. The following are some of the key characteristics of this service:

- ECS creates and manages the node cluster where containers are launched. The user needs to specify only the cluster size.
- Container health is monitored by container agents running on the node. The Container agent communicates to the master node that makes all service-related decisions. This allows for high availability of Containers.
- ECS takes care of scheduling the containers across the cluster. A scheduler API is implemented as a plugin and this allows integration with other schedulers such as Marathon and Kubernetes.
- ECS integrates well with other AWS services such as Cloudformation, ELB, logging, Volume management, and others.

Installing ECS and an example

ECS can be controlled from the AWS console or using the AWS CLI or ECS CLI. For the following example, I have used the ECS CLI, which can be installed using the procedure in this link (`http://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS_CLI_installation.html`).

I used the following example (`http://docs.aws.amazon.com/AmazonECS/latest/developerguide/ECS_CLI_tutorial.html`) to create a WordPress application with two containers (WordPress and MySQL) using the compose YML file.

The following are the steps:

1. Create an ECS cluster.
2. Deploy the application as a service over the cluster.
3. The cluster size or service size can be dynamically changed later based on the requirements.

The following command shows the running containers of the WordPress application:

```
ecs-cli ps
```

The following is the output of the preceding command:

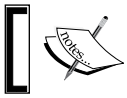
```
smakam14@jungle1:~/aws1$ ecs-cli ps
Name                                                State    Ports                     TaskDefinition
dff0a3a1-6ed1-4a62-94d6-50452cc16abd/mysql      RUNNING                            ecscompose-aws1:1
dff0a3a1-6ed1-4a62-94d6-50452cc16abd/wordpress  RUNNING  52.25.124.26:80->80/tcp   ecscompose-aws1:1
```

We can scale the application using the `ecs-cli` command. The following command scales each container to two:

```
ecs-cli compose --file hello-world.yaml scale 2
```

The following output shows the running containers at this point. As we can see, containers have scaled to two:

```
smakam14@jungle1:~/aws1$ ecs-cli ps
Name                                                State    Ports                     TaskDefinition
9f24df1b-bcb7-4666-a024-1e79e3e60566/wordpress  RUNNING  52.33.182.69:80->80/tcp   ecscompose-aws1:1
9f24df1b-bcb7-4666-a024-1e79e3e60566/mysql      RUNNING                            ecscompose-aws1:1
dff0a3a1-6ed1-4a62-94d6-50452cc16abd/mysql      RUNNING                            ecscompose-aws1:1
dff0a3a1-6ed1-4a62-94d6-50452cc16abd/wordpress  RUNNING  52.25.124.26:80->80/tcp   ecscompose-aws1:1
```



Note: The MySQL container is scaled typically using a single master and multiple slaves.

We can also log in to each AWS node and look at the running containers. The following output shows three containers in one of the AWS nodes. Two of them are application containers, and the third one is the ECS agent container that does container monitoring and talks to the master node:

```
[ec2-user@ip-10-0-0-208 ~]$ docker ps
CONTAINER ID        IMAGE                            COMMAND               CREATED             STATUS              PORTS
          NAMES
bed93cddc0b6        wordpress                        "/entrypoint.sh apac  About an hour ago   Up About an hour    0.0.0.0:80->80/tcp
          ecs-ecscompose-aws1-1-wordpress-d8c5a6e585d2e681f201
e8d81995684f        mysql                            "/entrypoint.sh mysq  About an hour ago   Up About an hour    3306/tcp
          ecs-ecscompose-aws1-1-mysql-8cd89cfcf6b6b7cad001
7b19fda216f3        amazon/amazon-ecs-agent:latest   "/agent"              2 hours ago         Up 2 hours          127.0.0.1:51678->51
678/tcp   ecs-agent
```

Note: To log in to each node, we need to use `ec2-user` as the username along with the private key used while creating the cluster.

To demonstrate HA, I tried stopping containers or nodes. Containers got rescheduled because the Container agent monitors containers in each node.

Google Container Engine

Google Container Engine is the cluster manager and container orchestration solution from Google that is built on top of Kubernetes. The following are the differences or benefits that we get from GCE compared to running a container cluster using Kubernetes as specified in the *Kubernetes installation* section:

- A node cluster is created automatically by Google Container engine. The user needs to specify only the cluster size and the CPU and memory requirement.
- Kubernetes is composed of multiple individual services such as an API server, scheduler, and agents that need to be installed for the Kubernetes system to work. Google Container engine takes care of creating the Kubernetes master with appropriate services and installing other Kubernetes services in agent nodes.
- Google Container engine integrates well with other Google services such as VPC networking, Logging, autoscaling, load balancing, and so on.
- The Docker hub, Google container registry, or on-premise registry can be used to store Container images.

Installing GCE and an example

The procedure at `https://cloud.google.com/container-engine/docs/before-you-begin` can be used to install the gcloud container components and kubectl. Containers can also be managed using the GCE dashboard.

I used the procedure at `https://cloud.google.com/container-engine/docs/tutorials/guestbook` to create a guestbook application containing three services. This application is the same as the one used in the *An example of Kubernetes application* section specified earlier.

The following are the steps:

1. Create a node cluster with the required cluster size. This will automatically create a Kubernetes master and appropriate agent services will be installed in the nodes.

2. Deploy the application using a replication controller and service files.
3. The cluster can be dynamically resized later based on the need.

The following is the cluster that I created. There are four nodes in the cluster as specified by `NUM_NODES`:

```
smakam14@jungle1:~$ gcloud container clusters list
NAME       ZONE           MASTER_VERSION  MASTER_IP     MACHINE_TYPE   NUM_NODES  STATUS
guestbook  us-central1-a  1.0.7           104.197.5.61  n1-standard-1  4          RUNNING
```

The following command shows the running services that consist of frontend, redis-master, and redis-slave. The Kubernetes service is also running in the master node:

```
Kubectl get services
```

The following is the output of the preceding command:

```
smakam14@jungle1:~$ kubectl get services
NAME           LABELS                                    SELECTOR                                  IP(S)            PORT(S)
frontend       app=guestbook,tier=frontend               app=guestbook,tier=frontend               10.127.252.184   80/TCP
                                                                                                   104.197.176.2
kubernetes     component=apiserver,provider=kubernetes   <none>                                    10.127.240.1     443/TCP
redis-master   app=redis,role=master,tier=backend        app=redis,role=master,tier=backend        10.127.244.163   6379/TCP
redis-slave    app=redis,role=slave,tier=backend         app=redis,role=slave,tier=backend         10.127.250.117   6379/TCP
```

As the frontend service is integrated with the GCE load balancer, there is also an external IP address. Using the external IP address, guestbook service can be accessed. The following command shows the list of endpoints associated with the load balancer:

```
Kubectl describe services frontend
```

The following is the output of the preceding command:

```
smakam14@jungle1:~$ kubectl describe services frontend
Name:                   frontend
Namespace:              default
Labels:                 app=guestbook,tier=frontend
Selector:               app=guestbook,tier=frontend
Type:                   LoadBalancer
IP:                     10.127.252.184
LoadBalancer Ingress:   104.197.176.2
Port:                   <unnamed>       80/TCP
NodePort:               <unnamed>       30578/TCP
Endpoints:              10.124.1.4:80,10.124.1.5:80,10.124.2.5:80
Session Affinity:       None
No events.
```

To resize the cluster, we need to first find the instance group associated with the cluster and resize it. The following command shows the instance group associated with the guestbook:

```
smakam14@jungle1:~$ gcloud container clusters describe guestbook --format yaml | grep -A 1 instanceG
roupUrls
instanceGroupUrls:
- https://www.googleapis.com/replicapool/v1beta2/projects/stunning-chain-108807/zones/us-central1-a/
instanceGroupManagers/gke-guestbook-e915700c-group
```

Using the instance group, we can resize the cluster as follows:

```
smakam14@jungle1:~$ gcloud compute instance-groups managed resize gke-guestbook-e915700c-group --zon
e us-central1-a --size 4
Updated [https://www.googleapis.com/compute/v1/projects/stunning-chain-108807/zones/us-central1-a/in
stanceGroupManagers/gke-guestbook-e915700c-group].
```

The initial set of outputs that show the cluster size as four were done after the resizing of the cluster.

We can log in to the individual nodes and see the containers launched in the node using regular Docker commands. In the following output, we see one instance of `redis-slave` and one instance of front end running in this node. Other Containers are Kubernetes infrastructure containers:

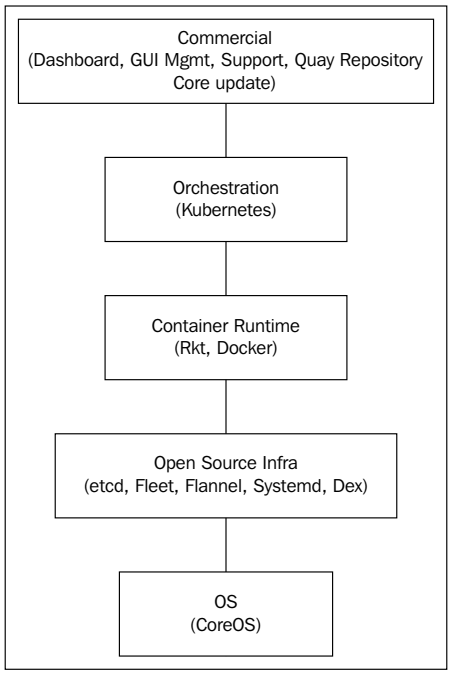
```
smakam14@gke-guestbook-e915700c-node-u5xw:~$ sudo docker ps
CONTAINER ID        IMAGE                                  COMMAND                  CREATED             STATUS              PORTS               N
AMES
a4353f82d1d2        gcr.io/google_samples/gb-redisslave:v1      "/entrypoint.sh /bin   13 hours ago        Up 13 hours                             k
8s_slave.b2cdceda_redis-slave-vbke8_default_3d41e543-8a2a-11e5-93d1-42010af00191_63af61e8
3e08690b9fb0        gcr.io/google_containers/pause:0.8.0        "/pause"                 13 hours ago        Up 13 hours                             k
8s_POD.49eee8c2_redis-slave-vbke8_default_3d41e543-8a2a-11e5-93d1-42010af00191_01518172
f9093a19e1aa        gcr.io/google_samples/gb-frontend:v3        "apache2-foreground"     13 hours ago        Up 13 hours                             k
8s_php-redis.6997cf0a_frontend-t01ov_default_14e39e84-8a2a-11e5-93d1-42010af00191_46a7f646
ed45376c8d9e        redis:latest                                "/entrypoint.sh redi   13 hours ago        Up 13 hours                             k
8s_master.3716a71d_redis-master-qvcuk_default_119c6d0f-8a2a-11e5-93d1-42010af00191_f1d4a89c
675a5aa7110d        gcr.io/google_containers/pause:0.8.0        "/pause"                 13 hours ago        Up 13 hours                             k
8s_POD.ef28e851_frontend-t01ov_default_14e39e84-8a2a-11e5-93d1-42010af00191_eaacae99
f07c8e744d41        gcr.io/google_containers/pause:0.8.0        "/pause"                 13 hours ago        Up 13 hours                             k
8s_POD.49eee8c2_redis-master-qvcuk_default_119c6d0f-8a2a-11e5-93d1-42010af00191_82979913
e1a86266d39d        gcr.io/google_containers/kube-ui:v1.1       "/kube-ui"               14 hours ago        Up 14 hours                             k
8s_kube-ui.bbb5835d_kube-ui-v1-wb11n_kube-system_892d5262-8a26-11e5-93d1-42010af00191_db2acff4
309bfa4e1837        gcr.io/google_containers/pause:0.8.0        "/pause"                 14 hours ago        Up 14 hours                             k
8s_POD.3b46e8b9_kube-ui-v1-wb11n_kube-system_892d5262-8a26-11e5-93d1-42010af00191_9c94cb9b
562b468841c3        gcr.io/google_containers/fluentd-gcp:1.11   "\"/bin/sh -c '/usr/   14 hours ago        Up 14 hours                             k
8s_fluentd-cloud-logging.44219385_fluentd-cloud-logging-gke-guestbook-e915700c-node-u5xw_kube-system_b845047be3634f41e2061ca65fbaa9d2_1a0e3d18
9da347cd4d82        gcr.io/google_containers/pause:0.8.0        "/pause"                 14 hours ago        Up 14 hours                             k
8s_POD.e4cc795_fluentd-cloud-logging-gke-guestbook-e915700c-node-u5xw_kube-system_b845047be3634f41e2061ca65fbaa9d2_e7fdba4e
```

CoreOS Tectonic

Tectonic is the commercial offering from CoreOS where they have integrated CoreOS and the open source components of CoreOS (Etcd, Fleet, Flannel, Rkt, and Dex) along with Kubernetes. With Tectonic, CoreOS is integrating their other commercial offerings such as CoreUpdate, Quay repository, and Enterprise CoreOS into Tectonic.

The plan is to expose the Kubernetes API as it is in Tectonic. Development in CoreOS open source projects will continue as it is, and the latest software will be updated to Tectonic.

The following diagram illustrates the different components of Tectonic:



Tectonic provides you with **Distributed Trusted Computing** (**DTM**), where security is provided at all layers including hardware and software. The following are some unique differentiators:

- At the firmware level, the customer key can be embedded, and this allows customers to verify all the software running in the system.
- Secure keys embedded in the firmware can verify the bootloader as well as CoreOS.
- Containers such as Rkt can be verified with their image signature.
- Logs can be made tamper-proof using the TPM hardware module embedded in the CPU motherboard.

Summary

In this chapter, we covered the importance of Container Orchestration along with the internals of popular container orchestration solutions, such as Kubernetes, Docker Swarm, and Mesos. There are many companies offering integrated Container orchestration solutions, and we covered a few popular ones such as the AWS Container service, Google Container Engine, and CoreOS Tectonic. For all the technologies covered in this chapter, installation and examples have been provided so that you can try them out. Customers have a choice of picking between integrated Container Orchestration solutions and manually integrating the Orchestration solution in their infrastructure. The factors affecting the choice would be flexibility, integration with in-house solutions, and cost. In the next chapter, we will cover OpenStack integration with Containers and CoreOS.

References

- The Kubernetes page: `http://kubernetes.io/`
- Mesos: `http://mesos.apache.org/` and `https://mesosphere.com/`
- Docker Swarm: `https://docs.docker.com/swarm/`
- Kubernetes on CoreOS: `https://coreos.com/kubernetes/docs/latest/`
- Google Container Engine: `https://cloud.google.com/container-engine/`
- AWS ECS: `https://aws.amazon.com/ecs/`
- Docker Compose: `https://docs.docker.com/compose`
- Docker machine: `https://docs.docker.com/machine/`
- Tectonic: `https://tectonic.com`
- Tectonic Distributed Trusted Computing: `https://tectonic.com/blog/announcing-distributed-trusted-computing/`

Further reading and tutorials

- Container Orchestration with Kubernetes and CoreOS: `https://www.youtube.com/watch?v=tA8XNVPZM2w`
- Comparing Orchestration solutions: `http://radar.oreilly.com/2015/10/swarm-v-fleet-v-kubernetes-v-mesos.html`, `http://www.slideshare.net/giganati/orchestration-tool-roundup-kubernetes-vs-docker-vs-heat-vs-terra-form-vs-tosca-1`, and `https://www.openstack.org/summit/vancouver-2015/summit-videos/presentation/orchestration-tool-roundup-kubernetes-vs-heat-vs-fleet-vs-maestrong-vs-tosca`
- Mesosphere introduction: `https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere`
- Docker and AWS ECS: 0`https://medium.com/aws-activate-startup-blog/cluster-based-architectures-using-docker-and-amazon-ec2-container-service-f74fa86254bf#.afp7kixga`

9
OpenStack Integration with Containers and CoreOS

OpenStack is an open source cloud operating system for managing public and private clouds. It is a pretty mature technology that is supported by the majority of the vendors and is used in a wide variety of production deployments. Running CoreOS in the OpenStack environment will give OpenStack users a Container-based Micro OS to deploy their distributed applications. Having Container orchestration integrated with OpenStack gives OpenStack users a single management solution to manage VMs and Containers. There are multiple projects ongoing in OpenStack currently to integrate Container management and Container networking with OpenStack.

The following topics will be covered in this chapter:

- An overview of OpenStack
- Running CoreOS in OpenStack
- Options to run Containers in OpenStack—the Nova Docker driver, Heat Docker plugin, and Magnum
- Container networking using OpenStack Kuryr and Neutron

An overview of OpenStack

Just like an OS for a desktop or server manages the resources associated with it, a cloud OS manages the resources associated with the cloud. Major cloud resources are compute, storage, and network. Compute includes servers and hypervisors associated with the servers that allows VM creation. Storage includes the local storage, **Storage Area Network** (**SAN**), and object storage.

Network includes vlans, firewalls, load balancers, and routers. A cloud OS is also responsible for other infrastructure-related items such as image management, authentication, security, billing, and so on. A cloud OS also provides some automated characteristics such as elasticity, a self service provisioning model, and others. Currently, the most popular open source cloud OS in the market is OpenStack. OpenStack has a lot of momentum going for it along with a great industry backing.

The following are some key OpenStack services:

- Nova: Compute
- Swift: Object storage
- Cinder: Block storage
- Neutron: Networking
- Glance: Image management
- Keystone: Authentication
- Heat: Orchestration
- Ceilometer: Metering
- Horizon: Web interface

OpenStack can be downloaded from `https://wiki.openstack.org/wiki/Get_OpenStack`. It is pretty complex to install OpenStack as there are multiple components involved. Similar to Linux distributions provided by Linux vendors, there are multiple vendors offering OpenStack distributions. The best way to try out OpenStack is using Devstack (`http://devstack.org/`). Devstack offers a scripted approach to install and can be installed on a laptop or VM. Devstack can be used to create a single-node cluster or multi-node cluster.

CoreOS on OpenStack

CoreOS can be run as a VM on OpenStack. CoreOS OpenStack images are available for alpha, beta, and stable versions.

Here, I have described the procedure to install CoreOS on OpenStack running in the Devstack environment. The procedure is based on the CoreOS OpenStack documentation (`https://coreos.com/os/docs/latest/booting-on-openstack.html`).

The following is a summary of the steps:

1. Get OpenStack Kilo running in Devstack. In my case, I installed Devstack in the Ubuntu 14.04 VM.
2. Set up the keys for authentication and a security group for SSH access.
3. Set up external network access and DNS for the VM. This is necessary as the CoreOS nodes need to discover each other using the token service.
4. Download the appropriate CoreOS image and upload to OpenStack using the Glance service.
5. Get a discovery token and update it in the user data configuration file.
6. Start CoreOS instances using custom user data specifying necessary services to be started and the number of instances to be started.

Get OpenStack Kilo running in Devstack

The following blog covers the procedure in detail:

`https://sreeninet.wordpress.com/2015/02/21/openstack-juno-install-using-devstack/`

This is the `local.conf` file that I used:

```
[[local|localrc]]
DEST=/opt/stack

# Logging
LOGFILE=$DEST/logs/stack.sh.log
VERBOSE=True
SCREEN_LOGDIR=$DEST/logs/screen
OFFLINE=True

# HOST
#EDITME
HOST_IP=<EDITME>

# Networking
FIXED_RANGE=10.0.0.0/24
disable_service n-net
enable_service q-svc
enable_service q-agt
enable_service q-dhcp
```

```
enable_service q-meta
enable_service q-l3
#ml2
Q_PLUGIN=ml2
Q_AGENT=openvswitch
# vxlan
Q_ML2_TENANT_NETWORK_TYPE=vxlan

# Credentials
ADMIN_PASSWORD=openstack
MYSQL_PASSWORD=openstack
RABBIT_PASSWORD=openstack
SERVICE_PASSWORD=openstack
SERVICE_TOKEN=tokentoken

#scheduler
enable_service n-sch
SCHEDULER=nova.scheduler.chance.ChanceScheduler

#vnc
enable_service n-novnc
enable_service n-cauth
```

Setting up keys and a security group

The following are the commands that I used to create a keypair and to expose port SSH and ICMP port of the VM:

```
nova keypair-add heattest > ~/Downloads/heattest.pem
nova secgroup-add-rule default icmp -1 -1 0.0.0.0/0
nova secgroup-add-rule default tcp 1 65535 0.0.0.0/0
```

Setting up external network access

The first command sets up the NAT rule for VM external access and the second command sets up a DNS server:

```
sudo iptables -t nat -A POSTROUTING -o eth0 -j MASQUERADE
neutron subnet-update  <subnet> --dns-nameservers list=true <dns address>
```

(Find `<subnet>` using `nova subnet-list` and `<dns address>` from the running host machine).

Download the CoreOS image and upload to Glance

The following command is used to download the latest alpha image and upload to OpenStack glance:

```
wget http://alpha.release.core-os.net/amd64-usr/current/coreos_
production_openstack_image.img.bz2
bunzip2 coreos_production_openstack_image.img.bz2
glance image-create --name CoreOS \
  --container-format bare \
  --disk-format qcow2 \
  --file coreos_production_openstack_image.img \
  --is-public True
```

The following is the `glance image-list` output and we can see the CoreOS image uploaded to Glance:

```
smakam14@sreeubuntu14-VirtualBox:~/devstack$ glance image-list
+--------------------------------------+---------------------------------+-------------+------------------+-----------+--------+
| ID                                   | Name                            | Disk Format | Container Format | Size      | Status |
+--------------------------------------+---------------------------------+-------------+------------------+-----------+--------+
| 5ec280dc-47a3-4376-8569-32055076bfff | cirros-0.3.4-x86_64-uec         | ami         | ami              | 25165824  | active |
| e06ee893-fdbf-4d7e-b850-a0cd0bbeab94 | cirros-0.3.4-x86_64-uec-kernel  | aki         | aki              | 4979632   | active |
| 14d06279-352a-41c4-8aca-04b3927056b8 | cirros-0.3.4-x86_64-uec-ramdisk | ari         | ari              | 3740163   | active |
| 8ae5223c-1742-47bf-9bb3-873374e61a64 | CoreOS                          | qcow2       | bare             | 670105600 | active |
+--------------------------------------+---------------------------------+-------------+------------------+-----------+--------+
```

Updating the user data to be used for CoreOS

I had some issues using the default user data to start CoreOS because there were issues with CoreOS determining the system IP. I raised a case (`https://groups.google.com/forum/#!topic/coreos-user/STmEU6FGRB4`) and the CoreOS team provided a sample user data where IP addresses are determined using a script inside the user data.

The following is the user data that I used:

```
#cloud-config

write_files:
  - path: /tmp/ip.sh
    permissions: 0755
    content: |
      #!/bin/sh
      get_ipv4() {
```

```
            IFACE="${1}"

            local ip
            while [ -z "${ip}" ]; do
                ip=$(ip -4 -o addr show dev "${IFACE}" scope global |
gawk '{split ($4, out, "/"); print out[1]}')
                sleep .1
            done

            echo "${ip}"
        }
        echo "IPV4_PUBLIC=$(get_ipv4 eth0)" > /run/metadata
        echo "IPV4_PRIVATE=$(get_ipv4 eth0)" >> /run/metadata

coreos:
  units:
    - name: populate-ips.service
      command: start
      runtime: true
      content: |
        [Service]
        Type=oneshot
        ExecStart=/tmp/ip.sh
    - name: etcd2.service
      command: start
      runtime: true
      drop-ins:
        - name: custom.conf
          content: |
            [Unit]
            Requires=populate-ips.service
            After=populate-ips.service

            [Service]
            EnvironmentFile=/run/metadata
            ExecStart=
            ExecStart=/usr/bin/etcd2 --initial-advertise-peer-
urls=http://${IPV4_PRIVATE}:2380 --listen-peer-urls=http://${IPV4_
PRIVATE}:2380 --listen-client-urls=http://0.0.0.0:2379 --advertise-
client-urls=http://${IPV4_PUBLIC}:2379 --discovery=https://discovery.
etcd.io/0cbf57ced1c56ac028af8ce7e32264ba
    - name: fleet.service
      command: start
```

The preceding user data does the following:

- The `populate-ips.service` unit file is used to update the IP address. It reads the IP manually and updates `/run/metadata` with the IP address.
- The discovery token is updated so that nodes can discover each other.
- Etcd2 service is started using the IP address set in `/run/metadata`.
- Fleet service is started using fleet unit file.

The following command is used to start two CoreOS instances using the preceding user data:

```
nova boot \
--user-data ./user-data1.yaml \
--image 8ae5223c-1742-47bf-9bb3-873374e61a64 \
--key-name heattest \
--flavor m1.coreos \
--num-instances 2 \
--security-groups default coreos
```

Note: For the CoreOS instance, I have used a custom flavor `m1.coreos` with 1 vcpu, 2 GB memory, and 10 GB hard disk. If these resource requirements are not met, instance creation will fail.

Let's look at the list of VMs. We can see the two CoreOS instances in the following image:

```
smakam14@sreeubuntu14-VirtualBox:~/devstack$ nova list
+--------------------------------------+----------+--------+------------+-------------+------------------+
| ID                                   | Name     | Status | Task State | Power State | Networks         |
+--------------------------------------+----------+--------+------------+-------------+------------------+
| c67e2b39-abfb-4429-a747-adbfbe1cf41d | coreos-1 | ACTIVE | -          | Running     | private=10.0.0.14 |
| 37793c82-b864-4ca0-9d91-549d061cc62d | coreos-2 | ACTIVE | -          | Running     | private=10.0.0.15 |
+--------------------------------------+----------+--------+------------+-------------+------------------+
```

The following command shows the CoreOS version running in OpenStack:

```
core@coreos-1 ~ $ cat /etc/os-release
NAME=CoreOS
ID=coreos
VERSION=845.0.0
VERSION_ID=845.0.0
```

The following command shows the etcd member list:

```
core@coreos-1 ~ $ etcdctl member list
887cae5451d131ec: name=20bca274e32f4ffda2247deed691ba2a peerURLs=http://10.0.0.14:2380 clientURLs=http://10.0.0.14:2379
9e9f9ae8de23bd31: name=c3e843ae3c5d4e5d9ef32367c67f291d peerURLs=http://10.0.0.15:2380 clientURLs=http://10.0.0.15:2379
```

The following command shows the fleet machines showing the two CoreOS nodes:

```
core@coreos-1 ~ $ fleetctl list-machines
MACHINE         IP              METADATA
20bca274...     10.0.0.14       -
c3e843ae...     10.0.0.15       -
```

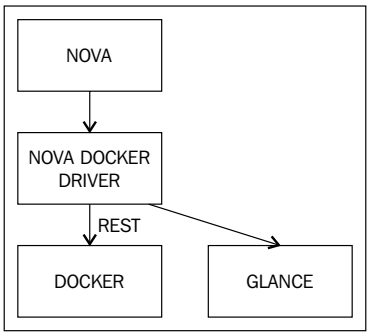
OpenStack and Containers

Even though OpenStack has supported VMs and baremetal for quite some time, Containers are pretty new to OpenStack. The initial focus in OpenStack was to extend VM Orchestration to also manage Containers. The Nova Docker driver and Heat Docker plugin are examples of this. This was not widely adopted as some of the Container functionality was missing in this approach. The OpenStack Magnum project addresses some of the limitations and manages Containers as a first-class citizen like a VM.

The Nova Docker driver

Nova typically manages VMs. In this approach, the Nova driver is extended to spawn Docker Containers.

The following diagram describes the architecture:



The following are some notes on the architecture:

- Nova is configured to use the Nova Docker driver for Containers
- The Nova Docker driver talks to the Docker daemon using the REST API
- Docker images are imported to Glance and the Nova Docker driver uses these images to spawn Containers

The Nova Docker driver is not present in the mainstream OpenStack installation and has to be installed separately.

Installing the Nova Driver

In the following example, we will cover the installation and usage of the Nova Docker driver to create Containers.

The following is a summary of the steps:

1. You need to have a Ubuntu 14.04 VM.
2. Install Docker.
3. Install the Nova docker plugin.
4. Do the stacking of Devstack.
5. Install nova-docker rootwrap filters.
6. Create Docker images and export to Glance.
7. Spawn Docker containers from Nova.

Installing Docker

The following is the Docker version running in my system after the Docker installation:

```
smakam14@sreeubuntu14-VirtualBox:~/devstack$ docker --version
Docker version 1.9.1, build a34a1d5
```

Install the Nova Docker plugin

Use the following command to install the plugin:

```
git clone -b stable/kilo https://github.com/stackforge/nova-docker.git
cd nova-docker
sudo pip install .
```

The following is the Docker driver version after installation:

```
smakam14@sreeubuntu14-VirtualBox:~/devstack$ sudo pip list | grep nova-docker
nova-docker (0.0.1.dev189)
```

The Devstack installation

I have used a stable Kilo release with the following `local.conf`. This sets up Nova to use the Docker driver:

```
[[local|localrc]]
# HOST
HOST_IP=<EDITME>

ADMIN_PASSWORD=openstack
DATABASE_PASSWORD=$ADMIN_PASSWORD
RABBIT_PASSWORD=$ADMIN_PASSWORD
SERVICE_PASSWORD=$ADMIN_PASSWORD
SERVICE_TOKEN=super-secret-admin-token
VIRT_DRIVER=novadocker.virt.docker.DockerDriver

# Logging
VERBOSE=True
DEST=$HOME/stack
SCREEN_LOGDIR=$DEST/logs/screen
SERVICE_DIR=$DEST/status
DATA_DIR=$DEST/data
LOGFILE=$DEST/logs/stack.sh.log
LOGDIR=$DEST/logs
OFFLINE=false

# Networking
FIXED_RANGE=10.0.0.0/24

# This enables Neutron
disable_service n-net
enable_service q-svc
enable_service q-agt
enable_service q-dhcp
enable_service q-l3
enable_service q-meta
```

```
# Introduce glance to docker images
[[post-config|$GLANCE_API_CONF]]
[DEFAULT]
container_formats=ami,ari,aki,bare,ovf,ova,docker

# Configure nova to use the nova-docker driver
[[post-config|$NOVA_CONF]]
[DEFAULT]
compute_driver=novadocker.virt.docker.DockerDriver
```

For installing the nova-docker rootwrap filters run the following command:

```
sudo cp nova-docker/etc/nova/rootwrap.d/docker.filters \
  /etc/nova/rootwrap.d/
```

For uploading the Docker image to Glance run the following command:

```
docker save nginx |  glance image-create --is-public=True --container-
format=docker --disk-format=raw --name nginx
```

Let's look at the Glance image list; we can see the nginx container image:

```
smakam14@sreeubuntu14-VirtualBox:~/devstack$ glance image-list | grep nginx
| d9d59e4c-f01c-46a5-9132-1a0295e2ddb9 | nginx                        | raw         | docker           | 139277824 | active |
```

Now, let's create the nginx container:

```
nova boot --flavor m1.small --image nginx nginxtest
```

Let's look at the Nova instances:

```
smakam14@sreeubuntu14-VirtualBox:~/devstack$ nova list
+--------------------------------------+-----------+--------+------------+-------------+------------------+
| ID                                   | Name      | Status | Task State | Power State | Networks         |
+--------------------------------------+-----------+--------+------------+-------------+------------------+
| 0881f120-b5c7-44c1-a8e1-c1f4e2132d1d | nginxtest | ACTIVE | -          | Running     | private=10.0.0.5 |
+--------------------------------------+-----------+--------+------------+-------------+------------------+
```

We can also see the running Container using the Docker native command:

```
smakam14@sreeubuntu14-VirtualBox:~/devstack$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
f51778a38d9e        nginx               "nginx -g 'daemon off"   About an hour ago   Up About an hour
81f120-b5c7-44c1-a8e1-c1f4e2132d1d                _
```

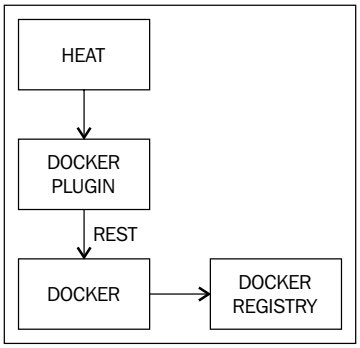
The Heat Docker plugin

The following are some of the items that the Nova Docker driver cannot do currently:

- Passing environment variables
- Linking containers
- Specifying volumes
- Orchestrating and scheduling the containers

These missing functionalities are important and unique for Containers. The Heat Docker plugin solves these problems partially, except for the orchestration part.

The following diagram shows the Heat Docker orchestration architecture:



The following are some notes on the architecture:

- Heat uses the Heat Docker plugin to talk to Docker. The Docker plugin uses the REST API to talk to the Docker engine.
- There is no direct interaction of Heat with the Docker registry.
- Using the Heat orchestration script, we can use all the features of the Docker engine. The disadvantage of this approach is that there is no direct integration of Docker with other OpenStack modules.

Installing the Heat plugin

I used the procedure at `https://sreeninet.wordpress.com/2015/06/14/openstack-and-docker-part-2/` and `https://github.com/MarouenMechtri/Docker-containers-deployment-with-OpenStack-Heat` to do the OpenStack Heat Docker plugin integration with OpenStack Icehouse.

Using the Heat plugin, we can spawn Docker containers either in the localhost or VM created by OpenStack.

I have used a Ubuntu 14.04 VM with Icehouse installed using Devstack. I used the procedure in the preceding links to install the Heat Docker plugin.

The following command output shows that the Heat plugin is successfully installed in the localhost:

```
$ heat resource-type-list | grep Docker
| DockerInc::Docker::Container
```

The following is a heat template file to spawn the nginx container in the localhost:

```
heat_template_version: 2013-05-23
description: >
  Heat template to deploy Docker containers to an existing host
resources:
  nginx-01:
    type: DockerInc::Docker::Container
    properties:
      image: nginx
      docker_endpoint: 'tcp://192.168.56.102:2376'
```

We have specified the endpoint as the localhost IP address and Docker engine port number.

The following command is used to create the Container using the preceding heat template:

```
heat stack-create -f ~/heat/docker_temp.yml nginxheat1
```

The following output shows that the heat stack installation is complete:

```
$ heat stack-list
+--------------------------------------+---------------+-----------------
+----------------------+
| id                                   | stack_name    | stack_status
| creation_time        |
+--------------------------------------+---------------+-----------------
+----------------------+
| d878d8c1-ce17-4f29-9203-febd37bd8b7d | nginxheat1    | CREATE_COMPLETE
| 2015-06-14T13:27:54Z |
+--------------------------------------+---------------+-----------------
+----------------------
```

The following output shows the successful running container in the localhost:

```
$ docker -H :2376 ps
CONTAINER ID        IMAGE               COMMAND                  CREATED
STATUS              PORTS               NAMES
624ff5de9240        nginx:latest        "nginx -g 'daemon of   2 minutes
ago       Up 2 minutes        80/tcp, 443/tcp     trusting_pasteur
```

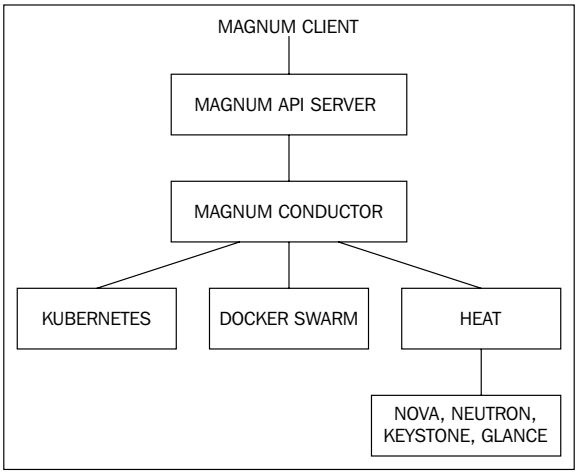
We can use the Heat plugin approach to run Containers on OpenStack VMs by changing the endpoint IP address from the localhost to the VM's IP address.

Magnum

With Nova Driver and Heat Orchestration, Containers were not a first-class citizen in OpenStack and Container specifics were not easy to manage with these approaches. Magnum is a generic Container management solution being developed in OpenStack to manage Docker as well as other Container technologies. Magnum supports Kubernetes, Docker Swarm, and Mesos for Orchestration currently. Other orchestration solutions will be added in the future. Magnum supports Docker Containers currently. The architecture allows it to support other Container runtime such as Rkt in the future. Magnum is still in the early stages and is available as a beta feature in the OpenStack Liberty release.

The Magnum architecture

The following diagram shows the different layers in Magnum:



The following are some notes on the Magnum architecture:

- The Magnum client talks to the Magnum API server, which in turn talks to the Magnum conductor. The Magnum conductor is responsible for interacting with Kubernetes, Docker Swarm, and Heat.
- Heat takes care of interacting with other OpenStack modules such as Nova, Neutron, Keystone, and Glance.
- Nova is used to create nodes in the Bay and they can run different Micro OSes such as CoreOS and Atomic.

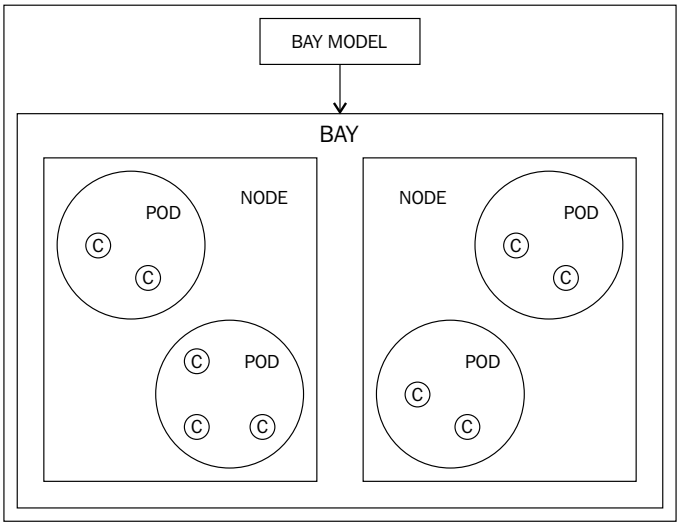
OpenStack Magnum uses the following constructs:

- **Bay model**: This is a cluster definition that describes properties of the cluster, such as the node flavor, node OS, and orchestration engine to be used. The following is an example bay model template that uses the node flavor as `m1.small`, fedora atomic as the base OS for the node, and Kubernetes as the orchestration engine:

```
magnum baymodel-create --name k8sbaymodel \
                       --image-id fedora-21-atomic-5 \
                       --keypair-id testkey \
                       --external-network-id public \
                       --dns-nameserver 8.8.8.8 \
                       --flavor-id m1.small \
                       --docker-volume-size 5 \
                       --network-driver flannel \
                       --coe kubernetes
```

- **Bay**: Bays are instantiated based on the bay model with the number of nodes necessary in Bay.
- **Nodes, Pods, and Containers**: Nodes are the individual VM instances. Pods are a collection of containers that share common properties and are scheduled together. Containers run within a Pod.

The following diagram shows the relationship between the Bay model, **Bay**, **Node**, **Pod**, and **Container**:



The following are the advantages of using OpenStack Magnum versus a native orchestration solution such as Kubernetes:

- For customers who are already using OpenStack, this provides an integrated solution.
- OpenStack provides multitenancy at all layers. This can be extended for Containers as well.
- OpenStack Magnum allows interaction with other OpenStack modules such as Neutron, Keystone, Glance, Swift, and Cinder. Some of these integrations are planned for the future.
- VMs and Containers have different purposes and most likely, they will coexist. OpenStack with the Magnum project provides you with an orchestration solution covering both VMs and Containers and this makes it very attractive.

Installing Magnum

Magnum can be installed using the procedure at `https://github.com/openstack/magnum/blob/master/doc/source/dev/quickstart.rst`. The following is a summary of the steps:

1. Create the OpenStack development environment with Devstack enabling the Magnum service.

2. By default, the Fedora Atomic image gets downloaded to Glance as part of the Devstack installation. If the CoreOS image is necessary, we need to download it manually to Glance.
3. Create a Bay model. A Bay model is like a template with a specific set of parameters using which multiple bays can be created. In the Bay model, we can specify the Bay type (currently supported Bay types are Kubernetes and Swarm), base image type (currently supported base images are Fedora Atomic and CoreOS), networking model (Flannel), instance size, and so on.
4. Create a Bay using the Bay model as a template. While creating a Bay, we can specify the number of nodes that need to be created. Node is a VM on top of which the base image is installed.
5. Deploy Containers using either Kubernetes or Swarm on top of the created Bay. Kubernetes or Swarm will take care of scheduling the Containers among the different nodes in the Bay.

Note: It is recommended that you avoid running Magnum in a VM. It is necessary to have a beefy machine as each Fedora instance requires at least 1 or 2 GB of RAM and 8 GB of hard disk space.

Container networking using OpenStack Kuryr

In this section, we will cover how Container networking can be done with OpenStack Neutron using the OpenStack Kuryr project.

OpenStack Neutron

OpenStack Neutron provides the networking functionality for OpenStack clusters. The following are some properties of OpenStack Neutron:

- Neutron provides networking as an API service with backends or plugins doing the implementation
- Neutron can be used for baremetal networking as well as VM networking
- Basic Neutron constructs are Neutron network, Port, Subnet, and Router
- Common Neutron backends are OVS, OVN, and Linux bridge
- Neutron also provides advanced networking services such as load balancing as a service, Firewall as a service, Routing as a service, and VPN as a service

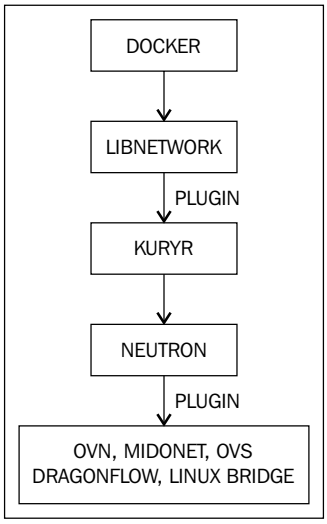
Containers and networking

We covered the details of Container networking in the earlier chapters. Some of the common technologies used were Flannel, Docker Libnetwork, Weave, and Calico. Most of these technologies use the Overlay network to provide Container networking.

OpenStack Kuryr

The goal of OpenStack Kuryr is to use Neutron to provide Container networking. Considering that Neutron is a mature technology, Kuryr aims to leverage the Neutron effort and make it easy for OpenStack users to adopt the Container technology. Kuryr is not a networking technology by itself; it aims to act as a bridge between Container networking and VM networking and enhancing Neutron to provide missing Container networking pieces.

The following diagram shows you how Docker can be used with Neutron and where Kuryr fits in:



The following are some notes on the Kuryr architecture:

- Kuryr is implemented as the Docker libnetwork plugin. Container networking calls are mapped by Kuryr to appropriate Neutron API calls.
- Neutron uses OVN, Midonet, and Dragonflow as backends to implement the Neutron calls.

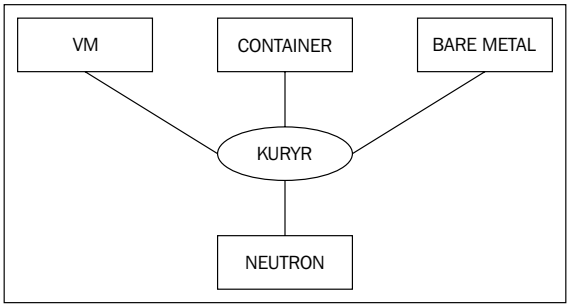
The following are some advantages of OpenStack Kuryr:

- It provides a common networking solution for both VMs and Containers.
- With Magnum and Kuryr together, Containers and VMs can have a common Orchestration.
- Considering that the Neutron technology is already mature, Containers can leverage all the Neutron functionalities.
- With default Container networking, there is a double encapsulation problem when Containers are deployed over a VM. Container networking does the first level of encapsulation and VM networking does the next level of encapsulation. This can cause performance overhead. With Kuryr, the double encapsulation problem can be avoided because Containers and VMs share the same network.
- Kuryr can integrate well with other OpenStack components to provide a complete Container solution with built-in multitenant support.

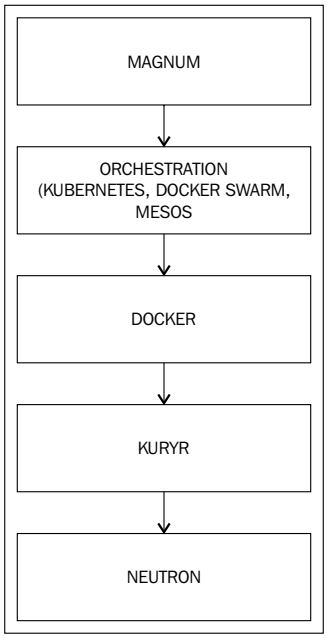
The following table shows the mapping between the Neutron and Libnetwork abstraction:

Neutron	**Libnetwork**
Neutron network	Network
Port	Endpoint
Subnet	IPAM
Plugin API (plug/unplug)	Plugin API (Join/leave)

The following diagram shows you how Kuryr can provide a common networking solution for Containers, VMs, and bare metal:



The following image shows you where Kuryr fits in with Magnum and Container orchestration projects:



The current state and roadmap of Kuryr

The Kuryr project is pretty new, and the Mitaka release will be the first OpenStack release with Kuryr support. The following are the ongoing and future work items with Kuryr:

- Adding missing Container features to Neutron, such as Port forwarding, resource tagging, and service discovery.
- Handling the nested container issue by integrating VM and Container networking.
- Better integration with OpenStack Magnum and Kolla projects.
- Current integration is focused on Docker. There are integration plans with the Kubernetes networking model.

Summary

In this chapter, we covered how Containers and CoreOS integrate with OpenStack. As CoreOS allows only applications running as Containers inside it, the OpenStack integration with CoreOS becomes more useful if OpenStack supports Container Orchestration. Even though the Nova driver and Heat plugin add Container support in OpenStack, the Magnum project seems like the correct solution treating Containers as a first-class citizen in OpenStack. We also covered how OpenStack Neutron can be used to provide Container networking using the Kuryr project. OpenStack Container integration is relatively new and there is still a lot of work ongoing to complete this integration. Managing VMs and Containers using single orchestration software gives tighter integration and eases the management and debugging capabilities. In the next chapter, we will cover CoreOS troubleshooting and debugging.

References

- Magnum: `https://wiki.openstack.org/wiki/Magnum`
- Magnum developer quick start: `https://github.com/openstack/magnum/blob/master/doc/source/dev/dev-quickstart.rst`
- CoreOS on OpenStack: `https://coreos.com/os/docs/latest/booting-on-openstack.html`
- The OpenStack Docker driver: `https://wiki.openstack.org/wiki/Docker`
- Installing Nova-docker with OpenStack: `http://blog.oddbit.com/2015/02/11/installing-novadocker-with-devstack/`
- OpenStack and Docker driver: `https://sreeninet.wordpress.com/2015/06/14/openstack-and-docker-part-1/`
- OpenStack and Docker with Heat and Magnum: `https://sreeninet.wordpress.com/2015/06/14/openstack-and-docker-part-2/`
- The OpenStack Heat plugin for Docker: `https://github.com/MarouenMechtri/Docker-containers-deployment-with-OpenStack-Heat`
- OpenStack Kuryr: `https://github.com/openstack/kuryr`
- OpenStack Kuryr background: `https://galsagie.github.io/sdn/openstack/docker/kuryr/neutron/2015/08/24/kuryr-part1/`

Further reading and tutorials

- Private Cloud Dream Stack - OpenStack + CoreOS + Kubernetes: `https://www.openstack.org/summit/vancouver-2015/summit-videos/presentation/private-cloud-dream-stack-openstack-coreos-kubernetes`
- Magnum OpenStack presentations: `https://www.youtube.com/watch?v=BM6nFH7G8Vc` and `https://www.youtube.com/watch?v=_ZbebTIaS7M`
- Kuryr OpenStack presentations: `https://www.openstack.org/summit/tokyo-2015/videos/presentation/connecting-the-dots-with-neutron-unifying-network-virtualization-between-containers-and-vms` and `https://www.youtube.com/watch?v=crVi30bgOt0`

10
CoreOS and Containers – Troubleshooting and Debugging

Both CoreOS and Containers pose some special challenges in troubleshooting and there are ways to overcome this problem. CoreOS, being a Container-optimized OS, does not support a package manager, and this prevents the installation of some of the Linux debugging tools. This can be overcome by running the Linux tools in a Container with a tool called **Toolbox** provided by CoreOS. Containers run in their own namespaces, and the regular Linux tools do not give enough information to debug Containers. This problem is solved by tools such as `cadvisor` and `sysdig`. Logging is another important tool to debug system-level issues, and there are a few vendors such as LogEntries trying to solve this problem for Containers.

In this chapter, we will cover the following topics:

- Using CoreOS Toolbox and other CoreOS utilities to debug the CoreOS system
- Monitoring a Container using `sysdig` and `cadvisor`
- Docker remote API support
- Docker logging drivers
- Using LogEntries to do central Container log monitoring

CoreOS Toolbox

As CoreOS does not support a package manager, it is difficult to install custom tools for debugging problems, such as tcpdump, strace, and others. CoreOS provides you with a toolbox script that can start a Ubuntu or Fedora container with system-level privileges on top of which we can run Linux system tools, such as tcpdump to monitor and debug the CoreOS host.

To start Toolbox, run `/usr/bin/toolbox` from the CoreOS shell.

The following process output in the CoreOS host system shows that Toolbox has started with system-level privileges:

```
core@core-01 ~ $ ps -eaf|grep toolbox
core      1490  1233  0 16:30 pts/0    00:00:00 /bin/bash /usr/bin/toolbox
root      1635  1490  0 16:34 pts/0    00:00:00 systemd-nspawn --directory=/var/lib/toolbox/core-fedora-lates
t --capability=all --share-system --bind=/:/media/root --bind=/usr:/media/root/usr --bind=/run:/media/root/ru
n --user=root
```

Toolbox by default uses the Fedora image. The following output shows you Fedora inside the Toolbox container:

```
[root@core-01 ~]# cat /etc/os-release
NAME=Fedora
```

Tcpdump is not present in the default Fedora image. I was able to install `tcpdump` using `yum` and monitor the `eth0` interface from inside the Toolbox container. This shows one example of how Toolbox can be used.

To change the default Linux image that CoreOS Toolbox uses, we can specify a custom image in `~/.toolboxrc`.

The following is an example `.toolboxrc`, where we are asking Toolbox to use a Ubuntu image:

```
TOOLBOX_DOCKER_IMAGE=ubuntu
TOOLBOX_DOCKER_TAG=14.04
```

If we start Toolbox after the preceding change, Toolbox will start a Ubuntu image with system-level privileges. The following is the Ubuntu image running as part of starting Toolbox:

```
root@core-01:~# cat /etc/os-release
NAME="Ubuntu"
```

We can specify the image selection in the `cloud-config` so that `.toolboxrc` is automatically written as part of the Container startup. The following is a sample `cloud-config` section where we specified `.toolboxrc` with Ubuntu as the default Toolbox Container image:

```
-write_files:
  - path: /home/core/.toolboxrc
    owner: core
    content: |
      TOOLBOX_DOCKER_IMAGE=ubuntu
      TOOLBOX_DOCKER_TAG=14.04
```

Other CoreOS debugging tools

We covered basic CoreOS debugging in *Chapter 2*, *Setting Up the CoreOS Lab* in the *Basic debugging* section. The following are a few utilities that can be used:

- The `journalctl` utility can be used to check the logs of all systemd services
- The `systemctl` utility can be used to check the status of all the services
- The `cloud-config` validator tool can be used to validate the `cloud-config` before using it with CoreOS
- Utilities such as Etcd, Fleet, Flannel, and Locksmith have their own debugging capabilities that can be turned on if necessary

Container monitoring

As Containers run in their own namespaces, traditional Linux monitoring tools such as top, ps, tcpdump, and lsof from the host system do not help monitor the activity happening within a Container or between Containers. This makes it complex to troubleshoot Containers. Before we discuss tools available for Container monitoring, let's see the major items that we need to monitor:

- The CPU usage by a Container and processes running inside a Container
- The memory usage by a Container and processes running inside a Container
- Network access for both incoming and outgoing connections
- File I/O performed by Containers

The following are some approaches to monitor Containers:

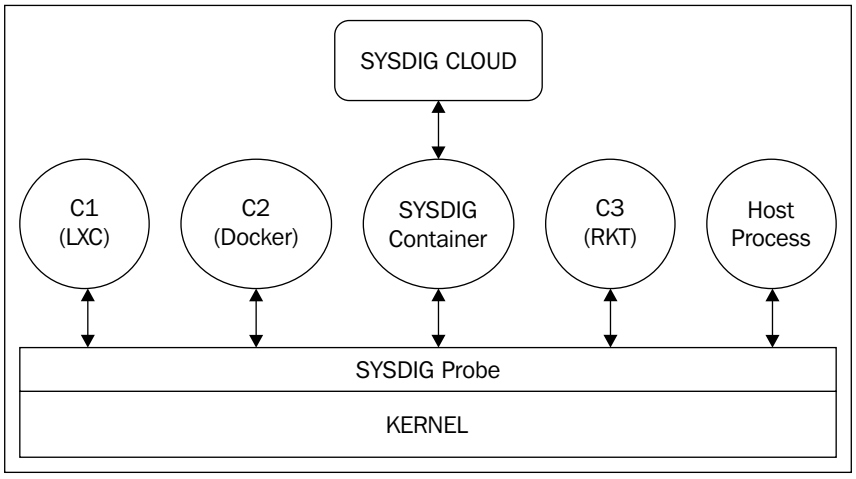
- Install monitoring software in the Container: This defeats the Container model where a Container runs a single microservice and is also not scalable.
- Install monitoring software in the host machine where the Container runs: This approach makes it difficult to install specialized software on a cluster OS like CoreOS as they allow only applications to run as Containers and not allow installing software in the base OS.
- Install monitoring software as a Container with system-level privileges: This is the most preferred approach.

Docker provides you with the `docker stats` command that provides basic CPU, memory, and I/O usage on a per Container basis. We covered `docker stats` in *Chapter 7*, *Container Integration with CoreOS – Docker and Rkt*. The data provided by Docker commands is very basic. There are many open source and commercial Container monitoring tools, such as cadvisor, sysdig, Data dog, newrelic, Prometheus, and Scout that provide more visibility in Containers. In this chapter, we will cover cadvisor and sysdig.

Sysdig

Sysdig is an open source project that provides Linux system-level visibility with built-in native support for Containers. Sysdig can be used for host monitoring as well as Container monitoring.

The following diagram shows the Sysdig architecture:



The following are some notes on the Sysdig architecture:

- Sysdig can monitor the host system, VM, and Containers.
- Sysdig can monitor different Container runtime like Docker, Rkt and LXC.
- The Sysdig documentation calls sysdig as `strace + tcpdump + htop + iftop + lsof + awesome sauce`.
- The Sysdig probe is a kernel module that needs to be installed in the host machine to do the monitoring. Sysdig has made the installation of this module very simple, and it works in regular Linux systems as well as in Container-based OSes, such as CoreOS and Rancher.
- Since sysdig directly monitors all Kernel system calls, sysdig provides more detailed monitoring data compared to other monitoring tools.
- A Sysdig container can be run on the host system and monitors the host processes as well as Containers running in the host system.
- Sysdig can monitor CPU, memory, network IO, and file IO. Sysdig provides various options to fine-tune the monitor query to provide relevant data.
- The Sysdig open source version has the sysdig CLI and csysdig, which has an ncurses-based interface. Csysdig is similar to htop, where we get an interactive text-based interface.
- The Sysdig cloud is the commercial version of Sysdig where data from multiple hosts and Containers are aggregated in a single location in the cloud and can be accessed as a SaaS application. The Sysdig cloud can be accessed from the cloud or installed on-premise.

Sysdig can be started as a Container. The following command shows you how to start the sysdig container:

```
docker run -i -t --name sysdig --privileged -v /var/run/docker.sock:/
host/var/run/docker.sock -v /dev:/host/dev -v /proc:/host/proc:ro sysdig/
sysdig
```

For more details on Sysdig installation, please refer `http://www.sysdig.org/install/`. The following command shows you a running sysdig container in a CoreOS system:

```
core@core-01 ~ $ docker ps | grep sysdig
ed3a44678ce0        sysdig/sysdig         "/docker-entrypoint.   40 minutes ago      Up 40 minutes                           sysdig
```

Examples of Sysdig

The following output shows you a list of Containers running in a CoreOS system on which we will try out some simple sysdig commands:

```
core@core-01 ~ $ docker ps
CONTAINER ID        IMAGE                  COMMAND                  CREATED             STATUS              PORTS                    NAMES
bd2479c0c1c9        wordpress              "/entrypoint.sh apac   11 minutes ago      Up 11 minutes       0.0.0.0:8080->80/tcp     wordpress

10e2c560b0ec        mysql                  "/entrypoint.sh mysq   11 minutes ago      Up 11 minutes       3306/tcp                 mysql

f957e6e598c3        nginx                  "nginx -g 'daemon of   11 minutes ago      Up 11 minutes       80/tcp, 443/tcp          nginx

c0f9c757ccf7        smakam/hellocounter    "python app.py"          11 minutes ago      Up 11 minutes       0.0.0.0:5000->5000/tcp   romantic_cray

ed3a44678ce0        sysdig/sysdig          "/docker-entrypoint.   11 minutes ago      Up 11 minutes                                sysdig

2e959ebeaf85        redis                  "/entrypoint.sh redi   13 minutes ago      Up 13 minutes       6379/tcp                 redis
```

The following command shows the top processes consuming the CPU. The output lists the PID in the host machine as well as the container. The `topprocs_cpu` utility is a chisel. In sysdig terms, each chisel is a script with some predefined task:

```
sysdig -pc -c topprocs_cpu
```

The following screenshot is the output of the preceding command:

```
CPU%             Process           Host_pid            Container_pid       container.name
--------------------------------------------------------------------------------
1.01%            python            1518                9                   romantic_cray
0.00%            bash              1207                1                   sysdig
0.00%            systemd-journal   381                 381                 host
0.00%            systemd-udevd     407                 407                 host
0.00%            systemd-resolve   554                 554                 host
0.00%            mysqld            1600                1                   mysql
0.00%            docker            675                 675                 host
0.00%            python            1510                1                   romantic_cray
0.00%            systemd           1                   1                   host
0.00%            sysdig            1939                190                 sysdig
```

The following command lists the top containers using network IO:

```
sysdig -pc -c topcontainers_net
```

The following screenshot is the output of the preceding command:

```
Bytes              container.name
----------------------------------------
341B               romantic_cray
69B                redis
```

The following command lists the top containers using file IO:

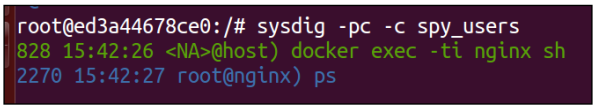
```
sysdig -c topcontainers_file
```

The following screenshot is the output of the preceding command:

The Sysdig spy command is useful to monitor all the external interactions to the host or Container. The following output shows the command executed in an nginx container when we used the exec command and performed `ps` in the container:

```
sysdig -pc -c spy_users
```

The following screenshot is the output of the preceding command:



The preceding output shows that the `docker exec` and `ps` commands were executed.

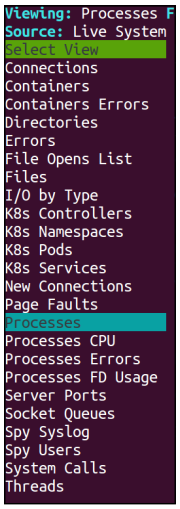
Csysdig

Csysdig is a text based UI for Sysdig. Csysdig is implemented as a customizable Curses UI. All operations that can be done through sysdig can also be done with csysdig. The csysdig user interface can be customized to show different views and the output can be filtered based on different user inputs.

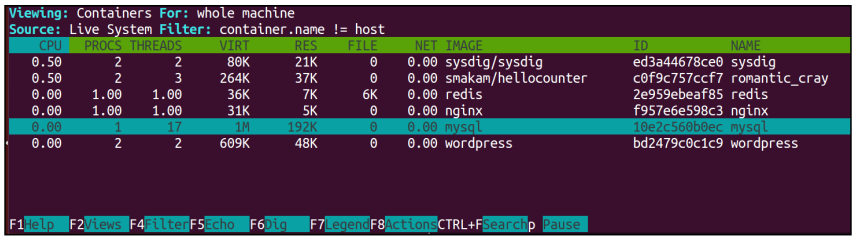
Csysdig can be started using the following command:

```
Cssysdig -pc (pc option gives container details)
```

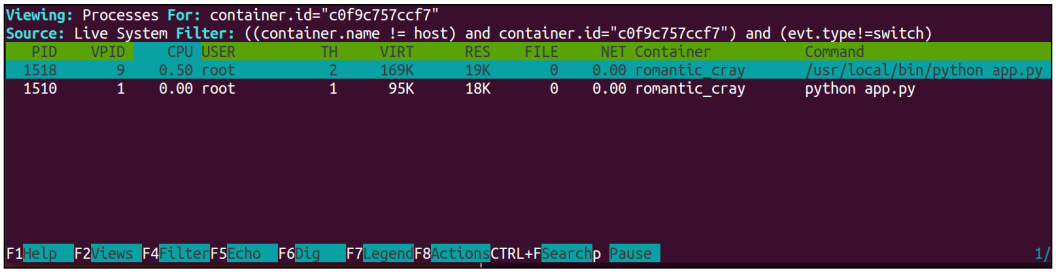
The following output shows different views possible in csysdig:



The following output lists the containers running in the host. This is available in the Container view:



Once we select a specific container, the following output shows the processes running in the container:



The Sysdig cloud

The sysdig cloud is a commercial solution from Sysdig where the sysdig data from the host machine is sent to a central server where the container and host monitoring data are collated from different hosts. The sysdig cloud can either be run on Sysdig's servers or as an on-premise solution.

The sysdig cloud is available on a 15-day trial period. I tried out the Sysdig cloud trial version and installed Sysdig in a CoreOS cluster running in AWS.

The following are the steps to install the Sysdig cloud and how to use it:

1. Register and create on online account in the Sysdig cloud. As part of registration, Sysdig will provide an access key.
2. The access key provided by Sysdig needs to be used in the host machine. Sysdig will use the access key to associate the hosts that are part of the same account.
3. When sysdig is started on the host machine, the sysdig agent will talk to the Sysdig server in the cloud and export the monitoring data.
4. The Sysdig cloud can also integrate with AWS. If we provide the AWS access key, Sysdig can automatically pull in AWS VM monitoring data as well.

The following is the CoreOS service unit file to start the sysdig-agent service in the host machine, which talks to the Sysdig cloud. The access key needs to be filled in appropriately. This unit starts the sysdig cloud agent in all the nodes of the CoreOS cluster as the `Global` option is set in X-Fleet:

```
[Unit]
Description=Sysdig Cloud Agent
After=docker.service
Requires=docker.service

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill sysdig-agent
ExecStartPre=-/usr/bin/docker rm sysdig-agent
ExecStartPre=/usr/bin/docker pull sysdig/agent
ExecStart=/usr/bin/docker run --name sysdig-agent --privileged --net
host --pid host -e ACCESS_KEY=<access key> -e TAGS=[role:web,location:
bangalore] -v /var/run/docker.sock:/host/var/run/docker.sock -v /dev:/
host/dev -v /proc:/host/proc:ro -v /boot:/host/boot:ro sysdig/agent
ExecStop=/usr/bin/docker stop sysdig-agent

[X-Fleet]
Global=true
```

The following is my three-node CoreOS cluster:

```
core@ip-172-31-30-52 /etc/systemd/system $ fleetctl list-machines
MACHINE         IP              METADATA
48ae5ab6...     172.31.30.51    -
62d3e36f...     172.31.30.52    -
ccd33655...     172.31.30.50    -
```

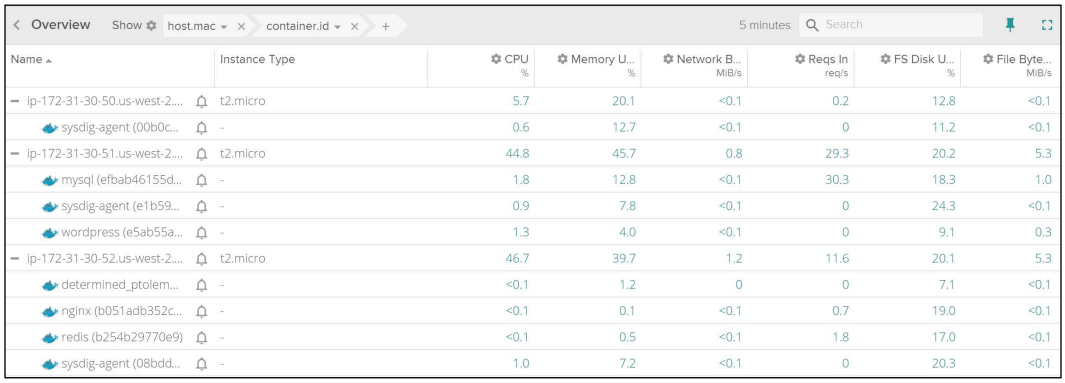
The following command can be used to start the sysdig agent on the CoreOS machine:

```
fleetctl start docker-sysdig.service
```

The following output shows the running sysdig-agent container in one of the CoreOS nodes:

```
core@ip-172-31-30-52 ~ $ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS               NAMES
08bdd4d987f5        sysdig/agent        "/docker-entrypoint.     55 minutes ago      Up 55 minutes                           sysdig-agent
```

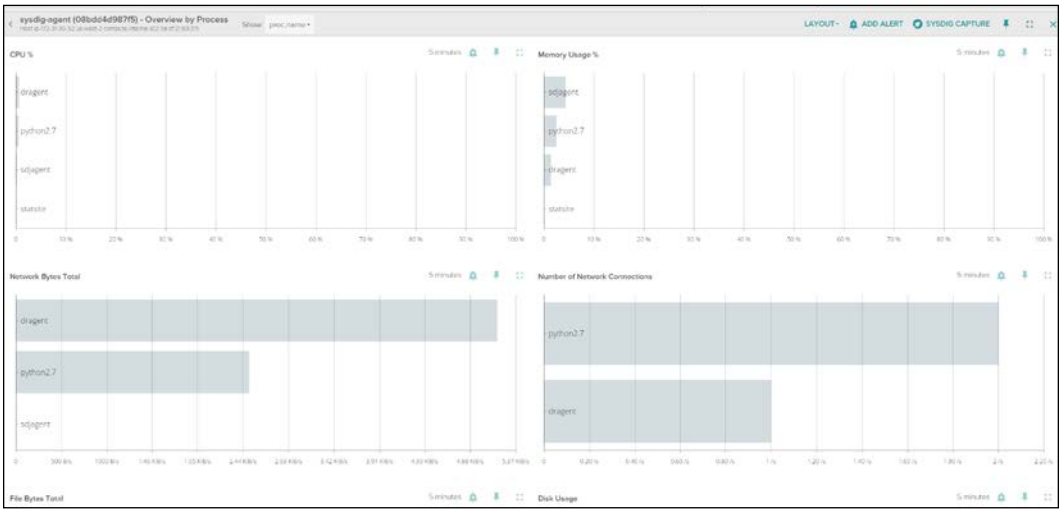
The following output in the Sysdig cloud shows the registered hosts along with the running containers. Here, we can see the three hosts and the Containers running in each host along with their CPU, memory, network IO, and file IO:



Name	Instance Type	CPU %	Memory U... %	Network B... MiB/s	Reqs In req/s	FS Disk U... %	File Byte... MiB/s
ip-172-31-30-50.us-west-2....	t2.micro	5.7	20.1	<0.1	0.2	12.8	<0.1
sysdig-agent (00b0c...	-	0.6	12.7	<0.1	0	11.2	<0.1
ip-172-31-30-51.us-west-2....	t2.micro	44.8	45.7	0.8	29.3	20.2	5.3
mysql (efbab46155d...	-	1.8	12.8	<0.1	30.3	18.3	1.0
sysdig-agent (e1b59...	-	0.9	7.8	<0.1	0	24.3	<0.1
wordpress (e5ab55a...	-	1.3	4.0	<0.1	0	9.1	0.3
ip-172-31-30-52.us-west-2....	t2.micro	46.7	39.7	1.2	11.6	20.1	5.3
determined_ptolem...	-	<0.1	1.2	0	0	7.1	<0.1
nginx (b051adb352c...	-	<0.1	0.1	<0.1	0.7	19.0	<0.1
redis (b254b29770e9)	-	<0.1	0.5	<0.1	1.8	17.0	<0.1
sysdig-agent (08bdd...	-	1.0	7.2	<0.1	0	20.3	<0.1

The following output shows the summary view:

The following output shows the dashboard output for a single Container with its associated processes. We have picked the sysdig container for the following output:



Kubernetes integration

Sysdig recently added a feature to integrate with Kubernetes, where Sysdig is aware of Kubernetes logical constructs, such as the master node, minion node, Pods, replication controllers, labels, and so on. Sysdig gets this awareness by querying the Kubernetes API server. By combining the data collected from Containers and the Kubernetes API server, Sysdig and the Sysdig cloud can group information at Kubernetes' level. For example, we can view the CPU and memory usage either on the Kubernetes pod or replication controller basis. Sysdig has plans to integrate with other orchestration engines such as Mesos and Swarm in the future. Sysdig also has plans to integrate with other Container runtime such as Rkt.

Cadvisor

Cadvisor is an open source tool from Google to monitor Containers as well as the host system on which the Container is running. Google developed cadvisor for its own Container system and later extended its support to Docker containers.

The following are some notes on cadvisor:

- It monitors CPU, memory, network, and file I/O for both the host system as well as Containers.
- It can work with Docker and other Container runtimes.
- It can be started as a Container in the host system with no special changes necessary in the host system.
- The cadvisor container starts a simple web server, using which we can access the dashboards using a simple GUI.
- It provides REST API for programmatic access.
- Cadvisor stores the history for only a small duration. It is necessary to use cadvisor with backends such as InfluxDB (`https://influxdata.com/`) and Prometheus (`https://prometheus.io/`) to maintain the history.

The following command can be used to start the Docker cadvisor Container:

```
docker run \
  --volume=/:/rootfs:ro \
  --volume=/var/run:/var/run:rw \
  --volume=/sys:/sys:ro \
  --volume=/var/lib/docker/:/var/lib/docker:ro \
  --publish=8080:8080 \
  --detach=true \
  --name=cadvisor \
  google/cadvisor:latest
```

The following output shows a running cadvisor container in the CoreOS system:

The following screenshot is a GUI snapshot showing processes and the CPU usage for a container:

The following output shows the REST API subtypes supported by cadvisor:

```
smakam14@jungle1:~$ curl -X GET http://172.17.8.102:8080/api/v1.3/
Supported request types: "containers,docker,events,machine,subcontainers"
```

The following are some examples of a REST API provided by cadvisor with the details that they provide. This link, `https://github.com/google/cadvisor/blob/master/docs/api.md`, gives the details of all the supported REST APIs. All the following commands return output in the JSON format.

The following command gives the host detail:

```
curl -X GET http://172.17.8.102:8080/api/v1.3/machine | jq .
```

The following command gives the Container performance detail:

```
curl -X GET http://172.17.8.102:8080/api/v1.3/containers/ | jq .
```

The following command gives the Docker container nginx performance detail:

```
curl -X GET http://172.17.8.102:8080/api/v1.3/docker/nginx | jq .
```

Cadvisor provides you with limited information compared to sysdig as cadvisor relies mainly on Docker-provided statistics. Additionally, cadvisor provides only limited history on statistics, and so it is necessary to integrate cadvisor with other tools such as Influxdb to maintain a longer history.

The Docker remote API

The Docker remote API can be used to access the Docker engine with the REST API. This can be used for programmatic access to Docker.

The following section of CoreOS `cloud-config` can be used to enable the Docker remote API listening on TCP port `2375`:

```
- name: docker-tcp.socket
  command: start
  enable: true
  content: |
    [Unit]
    Description=Docker Socket for the API
    [Socket]
    ListenStream=2375
    Service=docker.service
    BindIPv6Only=both
    [Install]
    WantedBy=sockets.target
```

The following are some examples of accessing the Docker remote API:

List the running Containers:

```
docker -H tcp://172.17.8.102:2375 ps
```

The following screenshot is the output of the preceding command:

```
core@core-01 ~ $ docker -H tcp://172.17.8.102:2375 ps
CONTAINER ID        IMAGE                COMMAND                  CREATED             STATUS              PORTS                    NAMES
ee3995120990        smakam/hellocounter  "python app.py"          7 minutes ago       Up 7 minutes        0.0.0.0:5000->5000/tcp   prickly_brown
7f871101f923        redis                "/entrypoint.sh redi   7 minutes ago       Up 7 minutes        6379/tcp                 redis
```

List the Container images:

The following command can be used to list Container images in the JSON format:

```
curl -X GET http://172.17.8.101:2375/images/json | jq .
```

The following screenshot is the output of the preceding command:

```
smakam14@jungle1:~$ curl -X GET http://172.17.8.102:2375/images/json | jq .
  % Total    % Received % Xferd  Average Speed   Time    Time     Time  Current
                                 Dload  Upload   Total   Spent    Left  Speed
100  3174    0  3174    0     0  44008      0 --:--:-- --:--:-- --:--:-- 44704
[
  {
    "Labels": null,
    "VirtualSize": 702646592,
    "Size": 0,
    "Created": 1448471644,
    "RepoDigests": [],
    "RepoTags": [
      "sysdig/agent:latest"
    ],
    "ParentId": "33b2702a9b492c39149655b7b3193f72df16c92b38fc567b3777a762069a209d",
    "Id": "4bec5fc7f1482a6998d93a5eaef32897e719c0341b14e5fcd13d8a3fac4b1b26"
  },
```

List Docker engine details:

The following command is equivalent to `docker info`:

```
curl -X GET http://172.17.8.101:2375/info | jq .
```

List particular Container statistics:

```
curl -X GET http://172.17.8.101:2375/containers/26b225ec6a8e/stats | jq .
```

List the Docker version:

```
curl -X GET http://172.17.8.102:2375/version | jq .
```

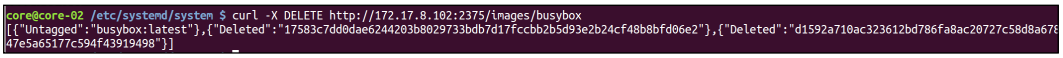
List the Docker events:

```
curl -X GET http://172.17.8.102:2375/events
```

The following command deletes the specific `busybox` container:

```
curl -X DELETE http://172.17.8.102:2375/images/busybox
```

The following screenshot is the output of the preceding command:



List specific Container logs sent to stdout:

The Container ID is specified as an argument for the following command:

```
curl -X GET http://172.17.8.101:2375/containers/5ab9abb4787e/
logs?stdout=1
```

If we need secure access to the Docker remote API, we can do it using TLS, and the Docker daemon supports this.

Container logging

When Containers send the output to stdout or stderr, it needs to be logged. This is useful to monitor errors and events and also to maintain the history of the Container application. With Containers, there are some special challenges with respect to logging:

- Typically, Containers run a microservice, and we don't want the logging process running inside a container as this defeats the Container model.
- With microservices, a single application can be split into multiple containers running across different hosts. It is necessary to collate logs from multiple containers to make meaningful conclusions. This enforces the need to have a central logging server rather than doing container monitoring on the host where the container is running.

We covered Container monitoring in the previous section. When Container logs are correlated with the Container monitoring data, we can get a better understanding of the system and easily narrow down any system wide issues.

I found the following two approaches widely used for centralized Container logging:

- ELK stack (Elastic search, Logstash, and Kibana): Elastic search is used as a central log repository, Logstash is used as an agent to export Container data, and Kibana is used as a logging GUI frontend. I have not covered the ELK stack in this chapter. The links in the references section provide details on setting up the ELK stack for Container logging.

- LogEntries: LogEntries combines the Container agent, frontend, and central logging server for a single integrated solution.

There are also other tools such as AWS Cloudwatch (`https://aws.amazon.com/cloudwatch/`), Loggly (`https://www.loggly.com`), Elastic (`http://www.elastic.io/`), and Sematext Logsene (`https://sematext.com/logsene/`) that provide logging capability for Containers. When using AWS cloudwatch for Container monitoring, we get custom hooks based on the AWS environment, and it also integrates well with their other cloud monitoring options.

Docker logging drivers

Docker supports the following log drivers as of Docker 1.7:

- `None`: No logging.
- `Json-file`: Logs are stored as a file in the JSON format. This is the default logging option.
- `Syslog`: Logs are sent to the syslog server.
- `Journald`: Logs are sent to the journald daemon. Journald is integrated with systemd.
- `Gelf`: This writes log messages to the GELF endpoint, such as Graylog or Logstash.
- `Fluentd`: This writes log messages to fluentd.
- `Awslogs`: This is the Amazon cloudwatch logging driver.

The JSON-file driver

The following command starts a Docker container with the json-file log driver with the maximum number of files limited to 100, each file not exceeding 1 MB:

```
docker run --name busyboxjsonlogger --log-driver=json-file --log-opt max-size=1m --log-opt max-file=100 -d busybox /bin/sh -c "while true; do echo hello world ; sleep 5 ; done"
```

In the preceding `busyboxjsonlogger` Container, we are continuously sending `hello world` output to stdout. The following output shows the `docker logs` output for `busyboxjsonlogger`, where we can see the `hello world` output:

```
core@core-01 ~ $ docker logs busyboxjsonlogger
hello world
hello world
```

The following command can be executed to find out the location of the json log file:

```
core@core-01 ~ $ docker inspect busyboxjsonlogger | grep LogPath
    "LogPath": "/var/lib/docker/containers/998bfcc3b78333d031be674f815f12fdf23161557dd00fef9d963b898f75327a/998bfcc3b78333d031be674f815f12fdf23161557dd00f
ef9d963b898f75327a-json.log",
```

Using the preceding path, we can directly dump the json logs, which gives additional information such as timestamp, stream type, and so on:

```
core@core-01 ~ $ sudo tail -f /var/lib/docker/containers/998bfcc3b78333d031be674f815f12fdf23161557dd00fef9d963b898f75327a/998bfcc3b78333d031be674f815f12fd
f23161557dd00fef9d963b898f75327a-json.log
{"log":"hello world\n","stream":"stdout","time":"2015-12-01T15:58:35.665472107Z"}
{"log":"hello world\n","stream":"stdout","time":"2015-12-01T15:58:40.665888844Z"}
```

The Syslog driver

The Syslog driver is useful to collate messages from multiple containers into a single server running the syslog daemon.

The following command can be used to start the syslog server as a container. This command exposes the syslog server to port `5514` in the host machine:

```
docker run -d -v /tmp:/var/log/syslog -p 5514:514/udp  --name rsyslog
voxxit/rsyslog
```

The following command can be used to start a container with the syslog driver option that sends the logs to the syslog server specified earlier:

```
docker run --log-driver=syslog --log-opt syslog-
address=udp://127.0.0.1:5514 --log-opt syslog-facility=daemon --log-opt
tag="mylog" --name busyboxsysloglogger -d busybox /bin/sh -c "while true;
do echo hello world ; sleep 5 ; done"
```

The following output shows the syslog from the syslog server:

```
docker exec rsyslog tail –f /var/log/messages
```

The following screenshot is the output of the preceding command:

```
core@core-01 ~ $ docker exec  rsyslog tail -f /var/log/messages
2015-12-01T16:10:59.378065+00:00 6f43dcc36804 rsyslogd: [origin software="rsyslogd" swVersion="8.9.0" x-pid="1" x-info="http://www.rsyslog.com"] start
2015-12-01T16:11:12Z core-01 docker/eacfe661f076[824]: hello world
2015-12-01T16:11:18Z core-01 docker/eacfe661f076[824]: hello world
```

The journald driver

The journald logging driver sends container logs to the systemd journal. Log entries can be retrieved using the `journalctl` command. This works well in a CoreOS environment where journald is used for all the other logging.

The following command starts a container with the journal driver:

```
docker run --name busyboxjournallogger --log-driver=journald -d busybox /
bin/sh -c "while true; do echo hello world ; sleep 5 ; done"
```

The following command shows the logs from journalctl with `CONTAINER_NAME` used as a filter:

```
journalctl CONTAINER_NAME=busyboxjournallogger
```

The following screenshot is the output of the preceding command:

```
core@core-01 ~ $ journalctl CONTAINER_NAME=busyboxjournallogger
-- Logs begin at Sun 2015-11-29 16:25:52 UTC, end at Tue 2015-12-01 16:41:19 UTC. --
Dec 01 16:40:40 core-01 docker[824]: hello world
Dec 01 16:40:45 core-01 docker[824]: hello world
```

The following command shows the journalctl logs in the JSON format:

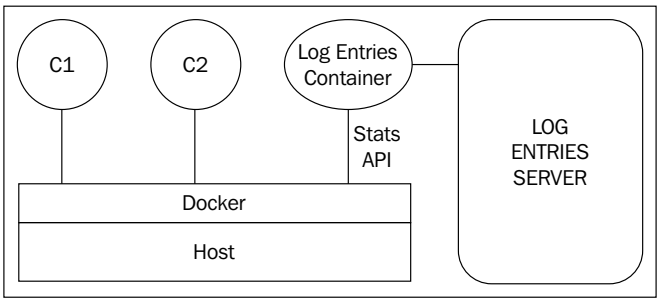
```
journalctl -o json CONTAINER_NAME=busyboxjournallogger --no-pager
```

The following screenshot is the output of the preceding command:

```
core@core-01 ~ $ journalctl -o json CONTAINER_NAME=busyboxjournallogger --no-pager
{ "__CURSOR" : "s=469ba1de85fe4817a1a0112544130064;i=15d7;b=e3fd093f625146d7a544b8abb07b2df4;m=30fc8a020;t=525d8d1705ad1;x=5ca1587b24b42514", "__REALTIME_
TIMESTAMP" : "1448988040518353", "__MONOTONIC_TIMESTAMP" : "13149708320", "_BOOT_ID" : "e3fd093f625146d7a544b8abb07b2df4", "PRIORITY" : "6", "_UID" : "0",
 "_GID" : "0", "_SYSTEMD_SLICE" : "system.slice", "_CAP_EFFECTIVE" : "3fffffffff", "_TRANSPORT" : "journal", "_MACHINE_ID" : "c8676a0252dc4053aec8ff0d0878
2d47", "_HOSTNAME" : "core-01", "_COMM" : "docker", "_EXE" : "/usr/bin/docker", "_CMDLINE" : "docker --daemon --host=fd:// --bip=10.1.51.1/24 --mtu=1472 -
-ip-masq=false", "_SYSTEMD_CGROUP" : "/system.slice/docker.service", "_SYSTEMD_UNIT" : "docker.service", "_PID" : "824", "MESSAGE" : "hello world", "CONTA
INER_ID" : "3515da606a49", "CONTAINER_ID_FULL" : "3515da606a49bc5eaeb7d87bf73cd57ed31f70fccf7e62f671bcf20690049d8d", "CONTAINER_NAME" : "busyboxjournallog
ger", "_SOURCE_REALTIME_TIMESTAMP" : "1448988040518092" }
```

Logentries

LogEntries can be used to collect logs from the host system running containers, export them to the central logging server, and analyze logs from a central server. The following diagram describes the components of the LogEntries Container architecture:



The following are some notes on the LogEntries Container architecture:

- The LogEntries container runs in the host system. It uses the Docker API to collect Container statistics, logs, and events, and transports them to a central server.
- The token-based system can be used to aggregate Container logs from multiple hosts. For a Container dataset belonging to a single domain, we can create a token from LogEntries and use this token in every individual host of the domain. The LogEntries agent in each host talks to the LogEntries server with this token. LogEntries aggregates log sets based on the token.
- As LogEntries collects Container statistics, it also displays some Container monitoring data in addition to logs.
- LogEntries provides extensions using Community packs. Community packs provide a way to share Search Queries, Tags, Alerts, and Widgets easily. Community packs follow a JSON structure and can be easily imported in the Logentries account via the LogEntries UI.
- LogEntries has both free and paid subscriptions. A paid subscription gives additional storage and enterprise-level features.

Exporting CoreOS journal logs

CoreOS uses journalctl to store logs from all services. The following Container (`https://github.com/kelseyhightower/journal-2-logentries`) can be used to send journal entries to the LogEntries server using SSL.

The following are the steps necessary to export journalctl logs from CoreOS nodes:

1. Create a token from logentries.
2. Use the token either in a service file or inside a `cloud-config` as an option while starting the `journal-2-logentries` container. An alternate option is to update the token in etcd, which all the nodes in a CoreOS cluster can use.
3. Update the token in etcd (for example, `etcdctl set /logentries.com/token <token>`).

The following service file can be used to start the `journal-2-logentries` container in all the CoreOS nodes of the cluster:

```
[Unit]
Description=Forward Systemd Journal to logentries.com

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill journal-2-logentries
```

```
ExecStartPre=-/usr/bin/docker rm journal-2-logentries
ExecStartPre=/usr/bin/docker pull quay.io/kelseyhightower/journal-2-
logentries
ExecStart=/usr/bin/bash -c \
"/usr/bin/docker run --name journal-2-logentries \
-v /run/journald.sock:/run/journald.sock \
-e LOGENTRIES_TOKEN=$(etcdctl get /logentries.com/token) \
quay.io/kelseyhightower/journal-2-logentries"

[X-Fleet]
Global=true
```

As the logentries container uses `journald.sock`, it is necessary to export that socket using the following unit in the `cloud-config`:

```
- name: systemd-journal-gatewayd.socket
  command: start
  enable: yes
  content: |
     [Unit]
     Description=Journal Gateway Service Socket
     [Socket]
     ListenStream=/var/run/journald.sock
     Service=systemd-journal-gatewayd.service
     [Install]
     WantedBy=sockets.target
```

The following output shows the `journal-2-logentries` service running in all the CoreOS nodes of the cluster:

```
core@core-01 ~ $ fleetctl list-units
UNIT                    MACHINE                         ACTIVE  SUB
logentries.service      5dfae98d.../172.17.8.101        active  running
logentries.service      c9024ea2.../172.17.8.103        active  running
logentries.service      feac3ccb.../172.17.8.102        active  running
```

The following output shows the `journal-2-logentries` Container running in one of the nodes:

```
core@core-01 ~ $ docker ps
CONTAINER ID        IMAGE                                       COMMAND               CREATED             STATUS              PORTS
          NAMES
30ebfd9db022        quay.io/kelseyhightower/journal-2-logentries   "/journal-2-logentri   2 hours ago         Up 2 hours
          journal-2-logentries
```

The following screenshot shows the LogEntries server frontend with journal logs from CoreOS nodes:

Container logs

LogEntries can be used to export Container logs, events, and statistics. Container events could be container start, create, stop, and die events. Container logs are the stdout and stderr logs. Container statistics are CPU, memory, file, and network IO related details.

The following are the steps necessary to export Container statistics and logs from CoreOS nodes:

1. Create a token from Logentries.
2. Use the token either in a service file or as an option while starting the docker-logentries container. An alternate option is to update the token in etcd, which all the nodes in the CoreOS cluster can use.
3. Update the token in etcd (for example, `etcdctl set /logentries.com/token <token>`).
4. To view Docker container statistics, it is necessary to use the Docker community pack. This is a JSON file and can be downloaded from `https://community.logentries.com/packs/`. The following instructions (`https://logentries.com/doc/community-packs/`) can be used to import the Docker community pack in logentries.

The following command can be used to start the `docker-logentries` container:

```
docker run -v /var/run/docker.sock:/var/run/docker.sock logentries/
docker-logentries -t <token>
```

The following service file can be used to start the `docker-logentries` container in all CoreOS nodes:

```
[Unit]
Description=Forward Forward Container logs/stats to logentries.com

[Service]
TimeoutStartSec=0
ExecStartPre=-/usr/bin/docker kill docker-logentries
ExecStartPre=-/usr/bin/docker rm docker-logentries
ExecStartPre=/usr/bin/docker pull logentries/docker-logentries
ExecStart=/usr/bin/bash -c \
"/usr/bin/docker run --name docker-logentries \
-v /var/run/docker.sock:/var/run/docker.sock \
-e LOGENTRIES_TOKEN=$(etcdctl get /logentries.com/token) \
logentries/docker-logentries"

[X-Fleet]
Global=true
```

The following output shows the `docker-logentries` service running on all CoreOS nodes:

```
core@core-01 ~ $ fleetctl list-units | grep docker
logentriesdocker.service        5dfae98d.../172.17.8.101        active  running
logentriesdocker.service        c9024ea2.../172.17.8.103        active  running
logentriesdocker.service        feac3ccb.../172.17.8.102        active  running
```

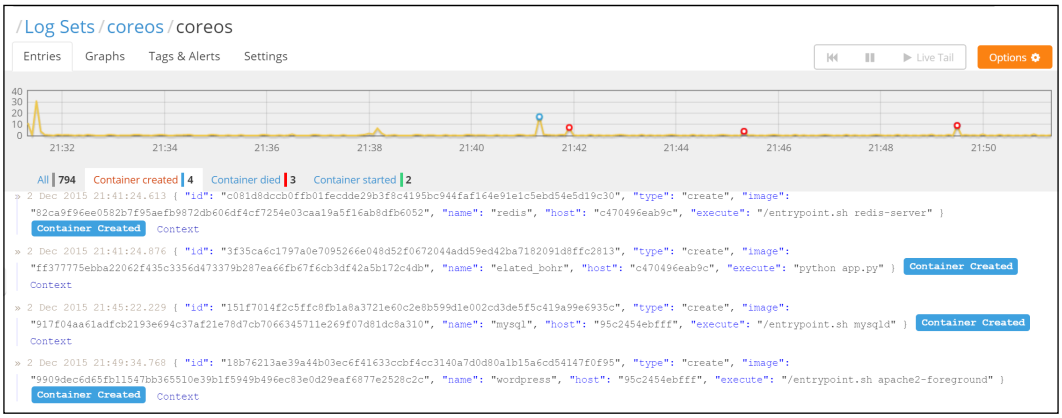
The following output shows the `docker-logentries` container running on one of the nodes:

```
core@core-01 ~ $ docker ps | grep docker
838cd0adc4bf        logentries/docker-logentries                "/usr/src/app/index.   8 minutes ago       Up 8 minutes
        docker-logentries
```

I created a bunch of Containers, and stopped and deleted a few to generate different Container events and logs.

The following output shows the Dashboard output that's received from the Docker community pack. The dashboard shows a summary of Container events along with Container monitoring data:

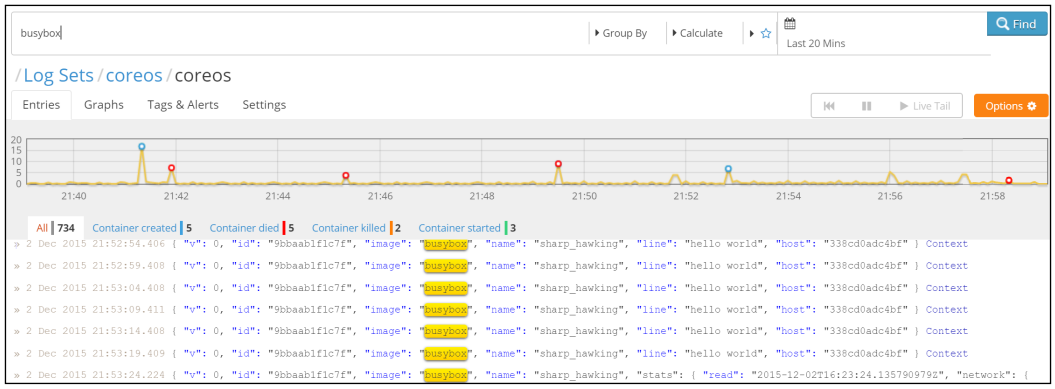
The following output shows the log set view, which also shows the specific containers created with each Container create event. In the following picture, we can see the create events for the Redis, WordPress, and MySQL containers:



To show the logging capability of LogEntries, I started the following container, which keeps sending `hello world` to stdout periodically:

```
docker run -d busybox /bin/sh -c "while true; do echo hello world; sleep
5; done"
```

The following output shows logs filtered by the `busybox` container name where we can see the stdout:



Summary

In this chapter, we covered ways to monitor and debug CoreOS systems as well as Docker Containers. Rather than approaching Containers and the host system as two separate entities, monitoring tools need to approach Containers and the host system as one entity and be able to provide both Container view as well as system view and correlate between the data. As Containers get deployed in hundreds and thousands across hosts, monitoring solutions need to be very scalable. There are a lot of developments happening with debugging and troubleshooting CoreOS systems and Docker Containers; multiple companies are trying to solve this problem. Companies such as Sysdig and Logentries have nice solutions to solve monitoring and logging problems. In the next chapter, we will cover production considerations for CoreOS, Docker Containers, and microservices.

References

- CoreOS Toolbox: `https://github.com/coreos/toolbox` and `http://thepracticalsysadmin.com/change-coreos-default-toolbox/`
- Cadvisor: `https://github.com/google/cadvisor`
- Comparing Container monitoring options: `http://rancher.com/comparing-monitoring-options-for-docker-deployments/`
- Sysdig: `https://sysdig.com/coreos-sysdig-part-1-digging-into-coreos-environments/`, `https://sysdig.com/sysdig-coreos-part-2-troubleshooting-flannel-networking-confd/`, `http://www.sysdig.org/`, and `https://github.com/draios/sysdig`

- Sysdig and Kubernetes integration: `https://sysdig.com/monitoring-kubernetes-with-sysdig-cloud/` and `https://sysdig.com/digging-into-kubernetes-with-sysdig/`
- Customizing the Docker remote API: `https://coreos.com/os/docs/latest/customizing-docker.html`
- The Docker logging driver: `http://docs.docker.com/engine/reference/logging/overview/`
- LogEntries: `https://logentries.com`
- Docker logging with ELK: `http://technologyconversations.com/2015/05/18/centralized-system-and-docker-logging-with-elk-stack/` and `http://evanhazlett.com/2014/11/Logging-with-ELK-and-Docker/`
- Docker logging with JSON and Syslog: `https://medium.com/@yoanis_gil/logging-with-docker-part-1-b23ef1443aac#.ehjyv77n7`

Further reading and tutorials

- Centralizing logs from a CoreOS cluster: `https://blog.logentries.com/2015/03/how-to-centralize-logs-from-coreos-clusters/`
- Docker logging enhancements with 1.7: `https://blog.logentries.com/2015/06/the-state-of-logging-on-docker-whats-new-with-1-7/`
- Logging on Docker webinar: `https://vimeo.com/123341629`
- The dark arts of Container monitoring: `https://www.youtube.com/watch?v=exna5ntTCpY`
- Sysdig and Logentries webinar: `https://www.youtube.com/watch?v=wNxteOCv5eE`
- Docker stats API: `https://blog.logentries.com/2015/02/what-is-the-docker-stats-api/` and `http://blog.scoutapp.com/articles/2015/06/22/monitoring-docker-containers-from-scratch`
- Sysdig Container visibility: `https://sysdig.com/let-light-sysdig-adds-container-visibility/`
- The Docker remote API: `http://blog.flux7.com/blogs/docker/docker-tutorial-series-part-8-docker-remote-api` and `http://blog.flux7.com/blogs/docker/docker-tutorial-series-part-9-10-docker-remote-api-commands-for-images`
- Protecting the Docker daemon: `https://docs.docker.com/engine/articles/https/`

11
CoreOS and Containers – Production Considerations

There is a big difference between running applications and containers in development versus production environments. Production environments pose a special set of challenges. The challenges mainly lie in scalability, high availability, security, and automation. CoreOS and Docker have solved significant challenges in taking applications from development to production. In this chapter, we will cover the production considerations for microservice infrastructure, including deployment, automation, and security.

The following topics will be covered in this chapter:

- CoreOS cluster design considerations
- Distributed infrastructure design consideration – Service discovery, deployment patterns, PaaS, and stateful and stateless Containers
- Security considerations
- Deployment and automation – CI/CD approaches and using Ansible for automation
- CoreOS and the Docker roadmap
- Microservice infrastructure – platform choices and solution providers

CoreOS cluster design considerations

The cluster size and update strategy are important design considerations for a CoreOS cluster.

The update strategy

The CoreOS automatic update feature keeps the nodes in the cluster secure and up-to-date. CoreOS provides you with various update mechanisms to control updates, and the user can select an approach based on their needs. We covered details of update strategies in *Chapter 3*, *CoreOS Autoupdate*. Some customers prefer doing the update only in the maintenance window and CoreOS gives control to do this.

Cluster considerations

The following are some considerations that need to be taken into account when choosing the CoreOS cluster. We have covered these individual topics in earlier chapters.

- Cluster size: A bigger cluster size provides better redundancy but updates take a little longer.
- Cluster architecture: We need to choose the architecture based on whether the cluster is used for development or production. For a production cluster, the preferable scheme is to have a small master cluster to run critical services such as Etcd and Fleet and have worker nodes point to the master cluster. Worker nodes should be used only to run application Containers.
- Etcd heartbeat and timeout tuning: These parameter values need to be tuned depending on whether the cluster is local or geographically distributed.
- Node backup and restore: Nodes can go bad. It is necessary to take periodic backups.
- Adding and removing nodes in the cluster: CoreOS provides mechanisms to add and remove nodes in the Etcd cluster dynamically without data loss. This can be used to grow the cluster size organically.

Distributed infrastructure design considerations

In this section, we will cover some miscellaneous infrastructure design considerations that were not covered in earlier chapters.

Service discovery

Microservices are dynamic and Service discovery refers to how microservices can find each other dynamically. Service discovery has three components:

- It discovers services automatically as they come up and accesses a service by the service name using DNS
- It maintains a shared database of services along with their access details that can be accessed from multiple hosts
- It accesses services using a load balancer and handles service failures automatically

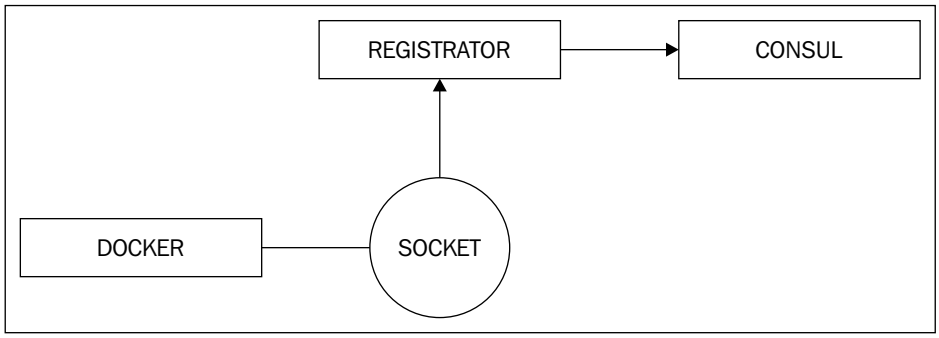
Service discovery is automatically taken care of when using a Container orchestration system such as Kubernetes. For smaller deployments, when there is no Orchestration system, we can do this manually using standalone tools.

We covered Service discovery in *Chapter 4, CoreOS Primary Services – Etcd, Systemd,* and *Fleet* in the *Service discovery* section using the Sidekick service and Etcd. This approach did not provide DNS lookup. The following approach is another way of doing service discovery with integrated DNS.

Service discovery using Registrator and Consul

Consul (`https://consul.io/`) and the Gliderlabs registrator (`https://github.com/gliderlabs/docker-consul/tree/consul-0.4`) in combination provide automatic service discovery and a service database.

The following figure shows the model:



The following points show you how this works:

- Consul provides service discovery, shared key-value storage, DNS-based service lookup and service health monitoring
- Gliderlabs registrator monitors the Docker socket for service creation and informs Consul about registration
- As DNS is integrated with Consul, services can be accessed by the service name

The following are the steps necessary to try this out in a Ubuntu Linux machine:

Set the Docker daemon to use the Docker bridge IP as one of the DNS lookup servers.

Add this line to `/etc/default/docker`:

```
DOCKER_OPTS="--dns 172.18.0.1 --dns 8.8.8.8 --dns-search service.
consul"
```

Restart the Docker daemon:

```
Sudo service docker restart
```

Start the Consul server:

```
docker run -d -p 8400:8400 -p 8500:8500 -p 172.18.0.1:53:8600/udp -h
node1 gliderlabs/consul-server -server –bootstrap
```

The preceding command exposes port `8400` for rpc, `8500` for UI, and `8600` for DNS. We mapped DNS to the Docker bridge IP address (`172.18.0.1`), and this allows us to access service names directly from inside Containers.

Start the Gliderlabs registrator:

```
docker run -d \
    --name=registrator \
    --net=host \
    --volume=/var/run/docker.sock:/tmp/docker.sock \
    gliderlabs/registrator:latest \
      consul://localhost:8500
```

In the preceding command, we also specified the location of Consul so that the registrator can register services to Consul.

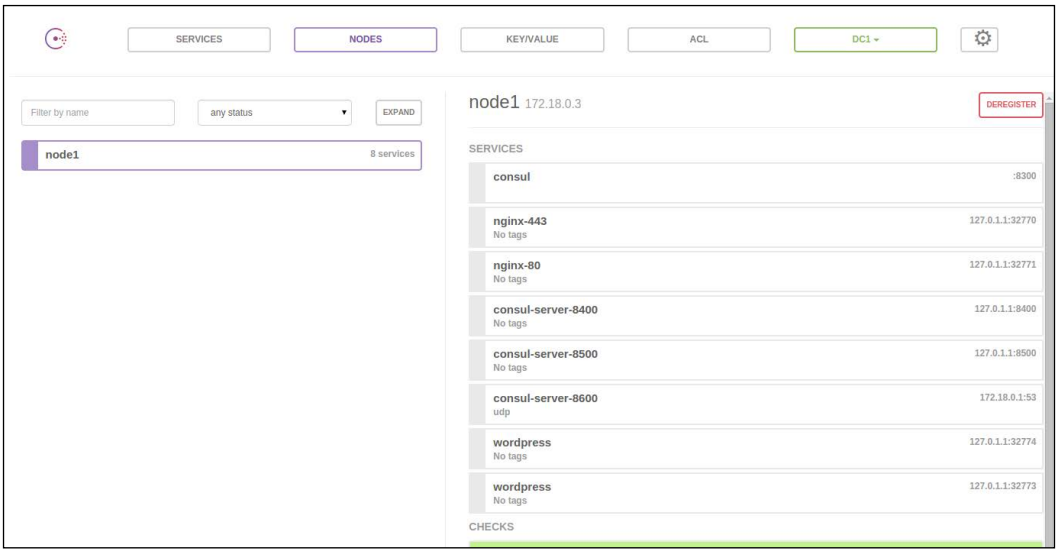
Now, let's start a few Containers:

```
docker run -d -P --name=nginx nginx
docker run --name mysql -e MYSQL_ROOT_PASSWORD=mysql -d mysql
docker run --name wordpress --link mysql:mysql -d -P wordpress
docker run --name wordpress1 --link mysql:mysql -d -P wordpress
```

Following output shows the running Containers:

```
smakam14@jungle1:~$ docker ps
CONTAINER ID        IMAGE                          COMMAND                  CREATED             STATUS              PORTS
                    NAMES
77a9919c64f2        wordpress                      "/entrypoint.sh apach"   25 minutes ago      Up 25 minutes       0.0.0.0:32774->80/tcp
                    wordpress1
be7659b2609f        wordpress                      "/entrypoint.sh apach"   25 minutes ago      Up 25 minutes       0.0.0.0:32773->80/tcp
                    wordpress
49ad22302de3        mysql                          "/entrypoint.sh mysql"   36 minutes ago      Up 36 minutes       3306/tcp
                    mysql
784003e3a4df        nginx                          "nginx -g 'daemon off"   36 minutes ago      Up 36 minutes       0.0.0.0:32771->80/tcp, 0.0.0
.0:32770->443/tcp
3c693aa39f2b        gliderlabs/registrator:latest  "/bin/registrator con"   36 minutes ago      Up 36 minutes
                    registrator
9d1074502e88        gliderlabs/consul-server       "/bin/consul agent -s"   45 minutes ago      Up 45 minutes       0.0.0.0:8400->8400/tcp, 8300
-8302/tcp, 8600/tcp, 8301-8302/udp, 0.0.0.0:8500->8500/tcp, 172.18.0.1:53->8600/udp   sad_thompson
```

We can look at the Consul UI to check whether the Services are registered. The Consul, NGINX, and WordPress Containers are seen in the following output along with their IP addresses and port numbers:



We can check whether the service lookup by the DNS name is working by accessing the service across Containers. The following output shows that the NGINX container is able to access the WordPress container by the service name, `wordpress`:

```
smakam14@jungle1:~$ docker exec -ti nginx sh
# ping -c1 wordpress
PING wordpress (172.18.0.5): 56 data bytes
64 bytes from 172.18.0.5: icmp_seq=0 ttl=64 time=0.230 ms
--- wordpress ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max/stddev = 0.230/0.230/0.230/0.000 ms
```

Dynamic load balancing

As part of Service discovery, a load balancer should be able to automatically find out active services and load balance among the active instances of the service. For example, when three instances of a web service are started, and if one of the instances dies, the load balancer should automatically be able to remove the inactive instance from the load balance list.

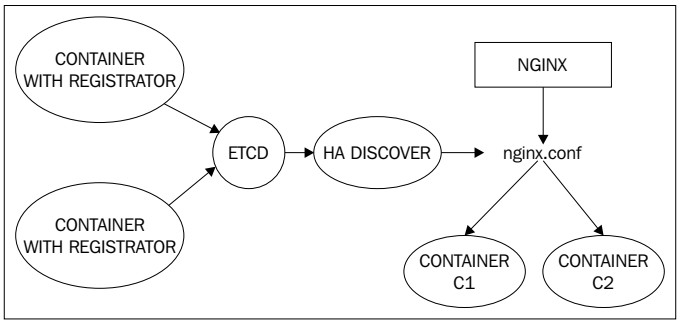
I found the following two approaches to be useful to achieve this.

Load balancing with confd and nginx

In the approach at `https://www.digitalocean.com/community/tutorials/how-to-use-confd-and-etcd-to-dynamically-reconfigure-services-in-coreos`, the following is a list of the steps:

1. The Sidekick service registers service details with `etcd`
2. Confd listens for etcd changes and updates `nginx.conf`
3. The Nginx load balancer does the load balancing based on entries in `nginx.conf`

The following diagram illustrates the load balancing with **ETCD**, **CONFD**, and **NGINX**:

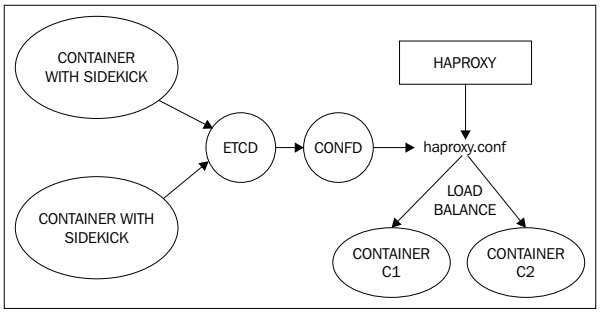


Load balancing with HAdiscover and HAproxy

In the approach at `http://adetante.github.io/articles/service-discovery-haproxy/`, the following is a list of the steps:

- The registrator registers the service details with `etcd`
- HAdiscover listens for changes to etcd and updates `haproxy.conf`
- HAproxy does the load balancing based on the HAproxy configuration

The following diagram illustrates the load balancing with ETCD, HA DISCOVER, and HAProxy.



Deployment patterns

We covered the advantages of microservices in the first chapter. Designing a microservice-based application is similar to object-oriented programming, where the Container image can be compared to a class and Containers can be compared to objects. There are many design patterns in object-oriented programming that specify how to split a monolithic application into classes and how classes can work together with other classes. Some of the object-oriented design principles also apply to microservices.

In *Chapter 8*, *Container Orchestration*, we covered Kubernetes Pods and how closely related containers can be grouped together in a single Pod. Design patterns such as Sidecar, ambassador, and adapter are pretty widely used to create Pods. Even though these design patterns are mentioned in the context of the Kubernetes pod, these can also be used in a non-Kubernetes-based system as well.

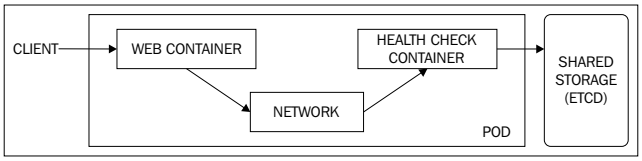
This link (`http://blog.kubernetes.io/2015/06/the-distributed-system-toolkit-patterns.html`) talks about common Kubernetes composite patterns.

The following are more details on common Kubernetes composite patterns.

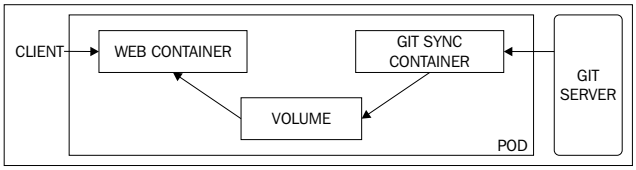
The Sidecar pattern

In the Sidecar pattern, there are two dependent Containers accomplishing a single task.

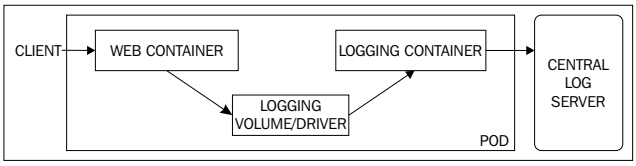
In the following diagram, the health check container monitors the web container and updates the results in a shared storage, such as **ETCD**, which can be used by the load balancer:



In the following diagram, the Git sync container updates data volume from the Git server, which is used by the web container to update the web page:



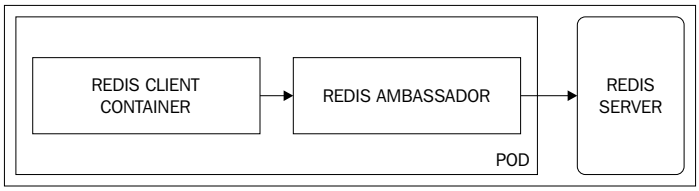
In the following diagram, the Web container updates the log volume that is read by the logging container to update the central log server:



The Ambassador pattern

The Ambassador pattern is used when there is a need to access different types of services from a client container and it is not efficient to modify the client container for each type of service. A proxy container will take care of accessing different types of service and the client container needs to talk only to the proxy container. For example, the redis proxy takes care of talking to a single redis master scenario or a scenario with a redis master and multiple redis slaves without the redis client being aware of the type of the redis service.

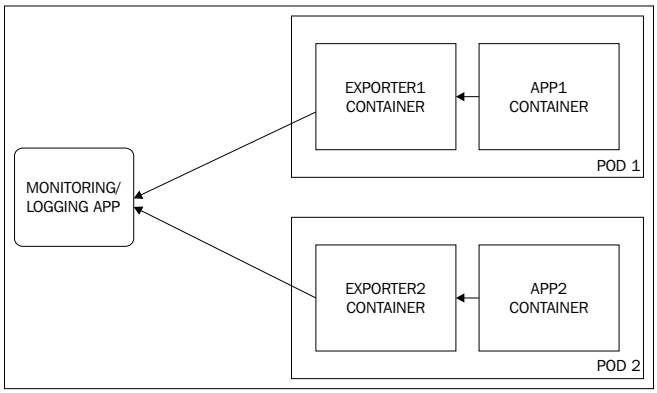
The following diagram shows the redis client with the redis ambassador accessing the redis service:



The Adapter pattern

The Adapter pattern is the inverse of the Ambassador pattern. An example of the Adapter pattern is a service container exposing a common interface independent of the application residing in the service. For example, a monitoring or logging application wants a common interface to gather inputs irrespective of the application type. An adapter container takes care of converting the data to a standard format expected by the monitoring or logging application.

The following example shows a monitoring/logging application accessing two different container applications, each with their own adapters:



Rolling updates with the Canary pattern

This is an upgrade approach used when an application runs as a Container across a cluster of servers behind a load balancer. In this approach, the application upgrade is done on a few servers, and based on the preliminary feedback from customers, the upgrade can either be continued or reverted.

Kubernetes supports a rolling upgrade with the Canary pattern. In the following example, we will demonstrate the Canary pattern with Kubernetes running on a CoreOS cluster in AWS. Here, we will upgrade `hello1-controller` with three replicas of the `hello:v1` container to `hello2-controller`, which also has three replicas of the `hello:v2` container.

For this example, we need a three-node Kubernetes CoreOS cluster. Installation instructions can be found in *Chapter 8*, *Container Orchestration*.

The following is a three-node cluster with one master and two worker nodes:

```
smakam14@jungle1:~/coreos-ops-tutorial$ kubectl get nodes
NAME                                        LABELS                                                    STATUS
ip-10-0-0-170.us-west-2.compute.internal    kubernetes.io/hostname=ip-10-0-0-170.us-west-2.compute.internal    Ready
ip-10-0-0-171.us-west-2.compute.internal    kubernetes.io/hostname=ip-10-0-0-171.us-west-2.compute.internal    Ready
```

The following is the replication controller, `hello1-controller.json`, with the `hello1` container image and three replicas:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: hello1
  labels:
    name: hello
spec:
  replicas: 3
  selector:
    name: hello
    version: v1
  template:
    metadata:
      labels:
        name: hello
        version: v1
    spec:
      containers:
      - name: hello
        image: quay.io/kelseyhightower/hello:1.0.0
        ports:
        - containerPort: 80
```

The following is the `hello-s.json` service using the `hello1` replication controller:

```
apiVersion: v1
kind: Service
metadata:
  name: hello
  labels:
    name: hello
spec:
  # if your cluster supports it, uncomment the following to
automatically create
  # an external load-balanced IP for the hello service.
  type: NodePort
  ports:
    # the port that this service should serve on
    - port: 80
  selector:
    name: hello
```

Let's start the replication controller and service:

```
kubectl create -f hello1-controller.json
kubectl create -f hello-s.json
```

Let's look at the running services and pods:

```
smakam14@jungle1:~/coreos-ops-tutorial$ kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)                              SELECTOR                REPLICAS
hello1       hello          quay.io/kelseyhightower/hello:1.0.0   name=hello,version=v1   3
smakam14@jungle1:~/coreos-ops-tutorial$ kubectl get services
NAME         LABELS                                    SELECTOR     IP(S)        PORT(S)
hello        name=hello                                name=hello   10.3.0.86    80/TCP
kubernetes   component=apiserver,provider=kubernetes   <none>       10.3.0.1     443/TCP
```

Let's create a new replication controller and perform a Canary pattern rolling upgrade. The following is the new replication controller, `hello2-controller.json`, using the `hello:2.0.0` container image:

```
apiVersion: v1
kind: ReplicationController
metadata:
  name: hello2
  labels:
    name: hello
spec:
  replicas: 3
  selector:
```

```
      name: hello
      version: v2
  template:
    metadata:
      labels:
        name: hello
        version: v2
    spec:
      containers:
      - name: hello
        image: quay.io/kelseyhightower/hello:2.0.0
        ports:
        - containerPort: 80
```

The following command does the rolling upgrade to `hello2`:

```
kubectl rolling-update hello1 --update-period=10s -f hello2-controller.
json
```

The update-period parameter specifies the time interval between the upgrade of each pod.

The following output shows you how each pod gets upgraded from `hello1` to `hello2`. At the end, the `hello1` replication controller is deleted:

```
smakam14@jungle1:~/coreos-ops-tutorial$ kubectl rolling-update hello1 --update-period=10s -f hello2-controller.json
Creating hello2
At beginning of loop: hello1 replicas: 2, hello2 replicas: 1
Updating hello1 replicas: 2, hello2 replicas: 1
At end of loop: hello1 replicas: 2, hello2 replicas: 1
At beginning of loop: hello1 replicas: 1, hello2 replicas: 2
Updating hello1 replicas: 1, hello2 replicas: 2
At end of loop: hello1 replicas: 1, hello2 replicas: 2
At beginning of loop: hello1 replicas: 0, hello2 replicas: 3
Updating hello1 replicas: 0, hello2 replicas: 3
At end of loop: hello1 replicas: 0, hello2 replicas: 3
Update succeeded. Deleting hello1
hello2
```

Let's look at the running replication controllers now. As we can see in the following output, `hello2` RC is running and `hello1` RC has been deleted:

```
smakam14@jungle1:~/coreos-ops-tutorial$ kubectl get rc
CONTROLLER   CONTAINER(S)   IMAGE(S)                          SELECTOR                REPLICAS
hello2       hello          quay.io/kelseyhightower/hello:2.0.0   name=hello,version=v2   3
```

Kubernetes also supports the rollback option. In case a problem is detected as part of a rolling upgrade, the rolling upgrade can be stopped and rollback can be done using the `--rollback` option.

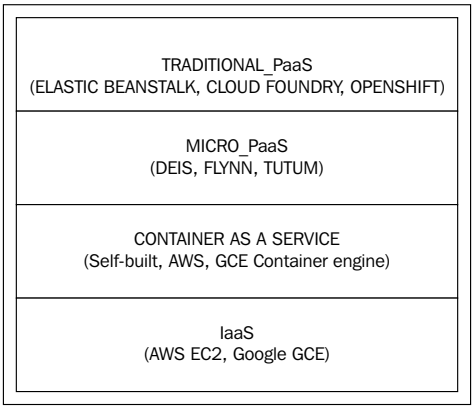
Containers and PaaS

Traditionally, the Services architecture has three types:

- **IaaS** (**Infrastructure as a service**)
- **PaaS** (**Platform as a service**)
- **SaaS** (**Software as a service**)

With the advent of Docker, the PaaS layer has become a little difficult to define. PaaS vendors have used Containers as their underlying technology from the beginning. In fact, Docker came from the Dotcloud Company, which was providing a PaaS service.

The following figure describes the new PaaS models and how they tie in to the traditional PaaS models and IaaS:



The following are some notes on the preceding diagram as well as how newer PaaS models are being developed:

- PaaS is typically used to simplify application deployment, which allows application developers to just develop the application, and PaaS provides necessary infrastructure services such as HA, scalability, and networking. PaaS is typically used for web applications.
- PaaS is typically deployed internally as Containers though users of PaaS need not be aware of this.
- Even though PaaS makes deploying applications faster, flexibility gets lost with PaaS.
- Examples of traditional PaaS systems are AWS Elastic beanstalk, Google GAE, Openshift, and Cloudfoundry.

- There is a new class of Micro-PaaS, where every service runs as a Docker Container, and this gives a little more flexibility than traditional PaaS. Examples are Deis, Flynn, and Tutum. Tutum was recently acquired by Docker.
- With Docker containers, Container orchestration systems such as Kubernetes, and Container OSes such as CoreOS, it becomes easier for customers to build a PaaS system by themselves, which gives them maximum flexibility. Both Amazon and Google have launched Container services where users can run their Containers. Users have the option to build Container services on top of their own infrastructure as well.

Stateful and Stateless Containers

Stateless containers are typically web applications such as NGINX, Node.js, and others. These follow the 12-factor application development (`http://12factor.net/`) methodology. These containers can be horizontally scaled. Stateful containers are used to store data like databases as data volumes in the host machine. Examples of stateful containers are Redis, MySQL, and MongoDB. We covered options for Container data persistence in *Chapter 6*, *CoreOS Storage Management*. When stateful containers are migrated, it is necessary to migrate the data associated with the stateful containers. The following options are available to migrate stateful containers:

- Using tools such as Flocker, which takes care of the volume and data migration when a Container moves across hosts
- Using a cluster-file system or NFS so that the same data volume can be seen across multiple hosts

If implementing stateful containers is difficult, the other option for storage is to keep databases separate from application containers and run them on special systems.

Security

The following are some approaches to secure the CoreOS cluster.

Secure the external daemons

Services such as Etcd, Fleet, and Docker can be reached externally. We can secure the client and server side using TLS and client and server certificates. We covered some of these details in earlier chapters when individual services were covered. If we are using Container orchestration such as Kubernetes, we need to make sure that the Kubernetes API server is using the TLS mechanism.

SELinux

SELinux is a Linux kernel feature that allows Container isolation even in case of a kernel bug that can cause the hacker to escape the Container namespace. SELinux integration is available from CoreOS 808.0 release. CoreOS disables SELinux by default. It can be enabled using the procedure at `https://coreos.com/os/docs/latest/selinux.html`. There are some limitations like not being able to run SELinux with the btrfs filesystem and with Containers sharing volumes.

Container image signing

Docker supports Container signing using the Docker content trust. Rkt supports image signing using GPG. Using these approaches, we can validate that Containers running on CoreOS come from reliable sources and the Container image is not tampered in the middle. Container image signing was covered in detail in *Chapter 7*, *Container Integration with CoreOS – Docker and Rkt*.

Deployment and automation

Containers make it easy to package and ship software and guarantee that the same Container can work in development as well as production environments. Combining Containers with good deployment and automation techniques will aid in faster software deployment.

Continuous Integration and Continuous Delivery

The traditional approach of releasing software has the following problems:

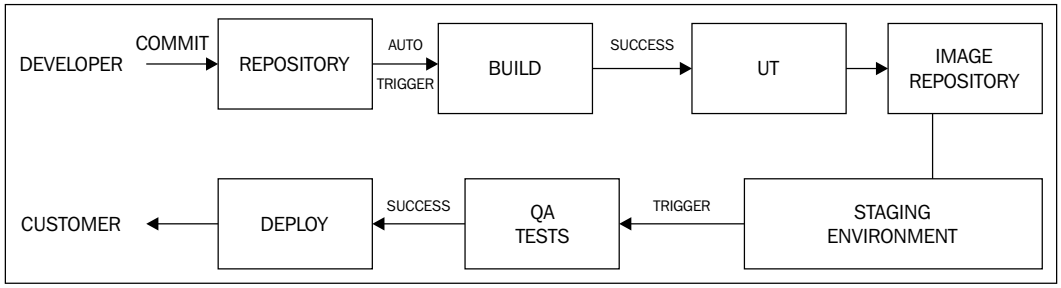
- Software release cycles were spaced apart, which caused new features taking a longer time to reach the customers
- Majority of the processes from the development stage to production were manual
- Considering the different deployment scenarios, it was difficult to guarantee that software worked in all environments and configurations

Containers have tried to mitigate some of these problems. Using microservices and the Container approach, it is guaranteed that the application will behave similarly in the development and production stages. Process automation and appropriate testing are still necessary for a Container-based environment.

Continuous Integration (**CI**) refers to the process of making an executable or Container image automatically after a developer has done the Unit testing and code commit.

Continuous Delivery (**CD**) refers to the process of taking the developer's built image, setting up the staging environment to test the image, and deploying it successfully for production.

The following figure illustrates the steps for CI/CD:



The following are some notes on the preceding diagram:

- The first row in the preceding diagram captures the steps for CI and the second row is the steps for CD.
- The CI process starts when developers commit the code after their basic UT. Typically, GitHub or Bitbucket is used as an image repository.
- There are hooks provided from the image repository to build system to automatically trigger the build after committing. The build system could be something such as Jenkins, which integrates with different code repositories. For Container images, `Dockerfile` and `docker build` will be used to build the Container image.
- Automatic UT suites can be kicked in if necessary before the image is committed to the image repository.
- The build itself needs to be done inside Containers in order to eliminate dependency on the host system.
- An image repository could be something such as the Docker hub or Docker trusted registry for Containers. CoreOS support the Quay repository for Container images.
- Once the image is pushed to a repository, the start of CD is automatically triggered.
- The staging environment needs to be set up with different Containers, storage, and other non-container software if necessary.

- QA tests are done in the staging environment. It is necessary that the staging environment be as close as possible to production.
- Once the QA tests are successful, images are deployed in production, such as AWS or GCE. If it's a PaaS application, it can be deployed to Cloudfoundry, among others.
- There are companies that provide integrated CI/CD solutions, such as Codeship, CircleCI, Shippable, and others. Docker has released an enterprise product called **Universal Control Plane** (**UCP**), which targets the CD part. Jenkins has Docker plugins to build images in Containers and also provides integration with the Docker hub.
- There are different deployment patterns to do the upgrade. We covered the Canary deployment pattern in an earlier section.

Ansible integration with CoreOS and Docker

Ansible is a configuration management and automation tool. Ansible is a very popular DevOps tool and serves similar purposes as Puppet or Chef. Ansible has a unique feature that there is no need to install an agent on the device side and this makes it very popular. There is active work ongoing to integrate Ansible with CoreOS and Docker. The following are some integration possibilities:

- Manage the CoreOS system with Ansible. As CoreOS does not come with Python installed and the fact that packages cannot be installed directly, there are some workarounds necessary to get Ansible to manage a CoreOS system.
- Ansible has a Docker module that simplifies Container management such as starting and stopping containers and controlling Container properties.
- The Docker installation can be automated with Ansible. Other than automating the Docker installation, Ansible can also manage other host infrastructure such as logging, storage, and networking.
- Ansible can be used to build Docker images instead of using Dockerfile. There is a `docker_image` module (`http://docs.ansible.com/ansible/docker_image_module.html`), but it is advised not to use it as its idempotent nature causes the Docker image to not be built in certain cases, which is a problem.

Using Ansible to manage CoreOS

I followed the procedure at `https://coreos.com/blog/managing-coreos-with-ansible/` to manage CoreOS with Ansible. As there is no package manager in CoreOS, Python cannot be installed directly. An approach at `https://github.com/defunctzombie/ansible-coreos-bootstrap` that is being used is to install PyPy, which is a minimal Python interpreter in CoreOS in the user directory and get Ansible to use this. The following example prepares the CoreOS node to be managed by Ansible and starts Etcd and Fleet service in the node using Ansible.

The following are the steps:

1. Install Ansible in the host machine. In my case, I am running Ansible 1.9 version in my Ubuntu 14.04 machine.
2. Create a CoreOS cluster.
3. Run the CoreOS bootstrap role to install the Python interpreter in CoreOS and update the system PATH to use it. Ansible roles create an abstraction over playbooks for specific tasks.
4. Run Ansible playbooks to start CoreOS services. Playbook is an Ansible task list.

Set up a CoreOS cluster:

The following commands set up the CoreOS cluster. In this case, a single-node cluster is created:

```
git clone https://github.com/defunctzombie/coreos-ansible-example.git
cd coreos-ansible-example
vagrant up
```

Set up passwordless SSH access:

Use the following command to set up passwordless SSH access. Ansible needs passwordless SSH access.

```
./bin/generate_ssh_config
```

Run the CoreOS bootstrap role:

The following command sets up a CoreOS node with Python using the Ansible role, `defunctzombie.coreos-bootstrap`:

```
ansible-galaxy install defunctzombie.coreos-bootstrap -p ./roles
ansible-playbook -i inventory/vagrant bootstrap.yml
```

I created the following playbook to start CoreOS services, Etcd2, and Fleet:

```
//Coreos_services.yml:
- name: CoreOS services
  hosts: web
  tasks:
    - name: Start etcd2
      service: name=etcd2.service state=started
      sudo: true
      sudo_user: root

    - name: Start fleet
      service: name=fleet.service state=started
      sudo: true
      sudo_user: root
```

In the preceding playbook, we have used the Ansible `service` module. Ansible modules are functions to do specific tasks. Ansible ships with a number of default modules and users can extend or write their own modules.

The following is the output when I started the playbook for the first time. The inventory file contains details of the single CoreOS node:

```
ansible-playbook –i inventory/vagrant coreos-services.yml
```

```
sreeni@ubuntu:~/coreos-ansible-example$ ansible-playbook -i inventory/vagrant coreos_services.yml

PLAY [CoreOS services] ********************************************************

GATHERING FACTS ***************************************************************
ok: [core-01]

TASK: [Start etcd2] ***********************************************************
changed: [core-01]

TASK: [Start fleet] ***********************************************************
changed: [core-01]

PLAY RECAP ********************************************************************
core-01                    : ok=3    changed=2    unreachable=0    failed=0
```

The following is the output when I ran the same playbook one more time:

```
sreeni@ubuntu:~/coreos-ansible-example$ ansible-playbook -i inventory/vagrant coreos_services.yml

PLAY [CoreOS services] ********************************************************

GATHERING FACTS ***************************************************************
ok: [core-01]

TASK: [Start etcd2] ***********************************************************
ok: [core-01]

TASK: [Start fleet] ***********************************************************
ok: [core-01]

PLAY RECAP ********************************************************************
core-01                    : ok=3    changed=0    unreachable=0    failed=0
```

As we can see, services don't get restarted as they have already started and the `changed` variable is not set.

The following output shows the running Etcd2 and Fleet services in the CoreOS node:

```
core@core-01 ~ $ systemctl list-units | grep "fleet\|etcd2"
etcd2.service                                                   loaded active running   etcd2
fleet.service                                                   loaded active running   fleet daemon
fleet.socket                                                    loaded active running   Fleet API Socket
```

Using Ansible to manage Docker Containers

Ansible provides you with a Docker module (`http://docs.ansible.com/ansible/docker_module.html`) to manage Docker Containers. The Docker module can manage the Container life cycle, which includes the starting and stopping of Containers. As Ansible modules are idempotent, we can use this functionality to pull Docker images only if necessary and restart Containers only if the base image has changed.

The following is a playbook that is executed on the same CoreOS node where we had run the CoreOS services playbook in the previous section. This will install the Docker module in the remote host and start the NGINX Container and a WordPress service having the WordPress and MySQL Containers:

```
//Coreos_containers.yml:
- name: CoreOS Container
  hosts: web
  tasks:
    - name: Install docker-py
      pip: name=docker-py version=1.1.0

    - name: pull container
      raw: docker pull nginx
```

```
- name: launch nginx container
  docker:
    image: "nginx"
    name: "example-nginx"
    ports: "8080:80"
    net: bridge
    state: reloaded

- name: launch mysql container
  docker:
    image: mysql
    name: mysql
    pull: always
    net: bridge
    state: reloaded
    env:
        MYSQL_ROOT_PASSWORD: mysql

- name: launch wordpress container
  docker:
    image: wordpress
    name: wordpress
    pull: always
    ports: 8000:80
    net: bridge
    state: reloaded
    links:
    - "mysql:mysql"
```

The following shows the output when the playbook is started for the first time:

```
sreeni@ubuntu:~/coreos-ansible-example$ ansible-playbook -i inventory/vagrant coreos_containers.yml

PLAY [CoreOS Container] ********************************************************

GATHERING FACTS ****************************************************************
ok: [core-01]

TASK: [Install docker-py] ******************************************************
ok: [core-01]

TASK: [pull container] *********************************************************
ok: [core-01]

TASK: [launch nginx container] *************************************************
changed: [core-01]

TASK: [launch mysql container] *************************************************
changed: [core-01]

TASK: [launch wordpress container] *********************************************
changed: [core-01]

PLAY RECAP *********************************************************************
core-01                    : ok=6    changed=3    unreachable=0    failed=0
```

The following screenshot shows the output when the same playbook is run again. As we can see, the `changed` flag is not set as all the Containers are running and there is no configuration change necessary:

```
sreeni@ubuntu:~/coreos-ansible-example$ ansible-playbook  -i inventory/vagrant coreos_containers.yml

PLAY [CoreOS Container] *******************************************************

GATHERING FACTS ***************************************************************
ok: [core-01]

TASK: [Install docker-py] *****************************************************
ok: [core-01]

TASK: [pull container] ********************************************************
ok: [core-01]

TASK: [launch nginx container] ************************************************
ok: [core-01]

TASK: [launch mysql container] ************************************************
ok: [core-01]

TASK: [launch wordpress container] ********************************************
ok: [core-01]

PLAY RECAP ********************************************************************
core-01                    : ok=6    changed=0    unreachable=0    failed=0
```

The following output shows the running Containers in the CoreOS node:

```
core@core-01 ~ $ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
       NAMES
3fc3ad2dd8f0        wordpress           "/entrypoint.sh apach"   5 minutes ago       Up 5 minutes        0.0.0.0:8000->80/tcp
       wordpress
0106d10a1653        mysql               "/entrypoint.sh mysql"   5 minutes ago       Up 5 minutes        3306/tcp
       mysql
a49842483587        nginx               "nginx -g 'daemon off"   6 minutes ago       Up 6 minutes        443/tcp, 0.0.0.0:8080->80
/tcp   example-nginx
```

Note: The `reloaded` flag in the Ansible Docker module should restart containers only if the base image is changed or configuration flags have changed. I hit a bug where Containers were restarted always. The link here (`https://github.com/ansible/ansible-modules-core/issues/1251`) describes this bug. Its workaround is to specify the `net` parameter as I have done in the preceding playbook.

The `reloaded` and `pull` flags are available from Ansible 1.9.

Ansible as a Container

Public Container images with Ansible preinstalled are available. This link, `https://hub.docker.com/r/ansible/ubuntu14.04-ansible/`, is an example Container image with Ansible preinstalled. The following output shows the Ansible version in the running Container:

```
sreeni@ubuntu:~$ docker run -ti ansible/ubuntu14.04-ansible bash
root@4e16dda3dcd0:/opt/ansible/ansible# ansible --version
ansible 2.0.0 (devel 1e50d31cdc) last updated 2015/10/22 18:29:22 (GMT +000)
```

Using Ansible to install Docker

Ansible has this concept of Roles that gives a good abstraction to share a list of playbooks that accomplish a single task. Ansible Roles are available to install Docker on the Linux host. Ansible Roles are maintained in a central repository called Ansible Galaxy, which can be shared across users. Ansible Galaxy is similar to the Docker hub for Ansible roles.

The following are the steps necessary:

1. Install the Ansible role locally from Ansible Galaxy.
2. Create a playbook with this role and run it.

I used this Galaxy role (`https://github.com/jamesdbloom/ansible-install-docker`) to install Docker on my Ubuntu node. There are a few other roles in Galaxy accomplishing the same task.

Use the following command to install the role:

```
ansible-galaxy install jamesdbloom.install-docker -p ./roles
```

Create the `install_docker1.yml` playbook with the role:

```
- name: install docker
  hosts: ubuntu
  gather_facts: True
  sudo: true
  roles:
    - jamesdbloom.install-docker
```

Run the playbook as follows:

```
ansible-playbook -i inventory/vagrant install_docker1.yml
```

The following is my inventory file:

```
## inventory file for vagrant machines
ubuntu-01 ansible_ssh_host=172.13.8.101

[ubuntu]
ubuntu-01

[ubuntu:vars]
ansible_ssh_user=vagrant
```

The following output shows the playbook output:

```
Ansible-playbook -i inventory/vagrant install_docker1.yml
```

```
sreeni@ubuntu:~/vagrant_ubuntu$ ansible-playbook -i inventory/vagrant install_docker1.yml

PLAY [install docker] *********************************************************

GATHERING FACTS ***************************************************************
ok: [ubuntu-01]

TASK: [jamesdbloom.install-docker | Install (or update) docker.io] ************
ok: [ubuntu-01]

TASK: [jamesdbloom.install-docker | Link docker binaries] *********************
ok: [ubuntu-01]

TASK: [jamesdbloom.install-docker | Check if docker installed] ****************
changed: [ubuntu-01]

TASK: [jamesdbloom.install-docker | Enable bash completion] *******************
changed: [ubuntu-01]

TASK: [jamesdbloom.install-docker | Expose docker host] ***********************
changed: [ubuntu-01]

NOTIFIED: [jamesdbloom.install-docker | restart docker] ***********************
changed: [ubuntu-01]

PLAY RECAP ********************************************************************
ubuntu-01                  : ok=7    changed=4    unreachable=0    failed=0
```

The following output shows Docker installed on my Ubuntu host using the preceding playbook:

```
vagrant@ubuntu-01:~$ sudo docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED             STATUS              PORTS
     NAMES
vagrant@ubuntu-01:~$ docker --version
Docker version 1.6.2, build 7c8fca2
```

Note: I faced an issue with restarting the Docker service. I was able to solve it using the procedure at `https://github.com/ansible/ansible-modules-core/issues/1170`, where the init file has to be removed. I faced this issue with Ansible 1.9.1; however, this is fixed in later Ansible versions.

The CoreOS roadmap

Ignition

The Ignition (`https://github.com/coreos/ignition`) project is being developed to setup initial CoreoS filesystem and it overcomes some of the issues with `coreos-cloudinit`. The `coreos-cloudinit` program is used to set up an initial CoreOS system configuration. The following are some known issues with `coreos-cloudinit`:

- It is difficult to feed in dynamic environment variables. This makes it difficult to run CoreOS in OpenStack environments and other environments where it is difficult to determine the IP address. This link, `https://groups.google.com/forum/#!topic/coreos-user/STmEU6FGRB4`, describes the case where IP addresses don't get set because of which `cloud-config` services fail in OpenStack.
- The `cloud-config` service is processed serially and we cannot specify dependencies.

Ignition is run once on initial system bring-up and it writes the necessary files like service files and configuration data. On the first boot, Ignition reads the configuration from a specific location that's specified in the bootloader.

Systemd, as part of a running provider metadata service file, will create `coreos-metadata.target`, which will contain necessary environment variables that service files can use. Service files will specify this target file as a dependency and systemd will take care of this dependency.

The following is a sample `etcd2.service` file, which specifies `coreos-metadata.service` as a dependency. The `/run/metadata/coreos` environment file will contain `COREOS_IPV4_PUBLIC`, and this will be generated by `coreos-metadata.service`:

```
[Unit]
Requires=coreos-metadata.service
After=coreos-metadata.service
```

```
[Service]
EnvironmentFile=/run/metadata/coreos
ExecStart=
ExecStart=/usr/bin/etcd2 \
    --advertise-client-urls=http://${COREOS_IPV4_PUBLIC}:2379 \
    --initial-advertise-peer-urls=http://${COREOS_IPV4_LOCAL}:2380 \
    --listen-client-urls=http://0.0.0.0:2379 \
    --listen-peer-urls=http://${COREOS_IPV4_LOCAL}:2380 \
    --initial-cluster=${ETCD_NAME}=http://${COREOS_IPV4_LOCAL}:2380
```

Ignition will be backward-compatible with cloudinit. Ignition has not yet been officially released.

DEX

DEX is an open source project started by CoreOS for identity management, including authentication and authorization. The following are some properties of DEX:

- DEX uses the **OpenID connect** (**OIDC**) (`http://openid.net/connect/`) standard, which is built on OAuth 2.0. OAuth 2.0 is used by Google to sign in to their services such as Gmail.
- DEX supports multiple identity providers using the Connectors module. Currently, DEX supports the local connector using local servers and a OIDC connector such as Google.
- There is a plan to add authorization, user management, and multiple other connectors such as LDAP and GitHub.
- DEX is used as an identity provider in the Tectonic project.

DEX is still in its early stages and under active development.

Clair

Clair is an open source project started by CoreOS to detect Container vulnerabilities. The following are some properties of Clair:

- Clair scans Container images stored in the Quay Container repository for vulnerabilities
- Each Container layer contains information about packages installed in that layer and this is provided by the corresponding Linux package manager

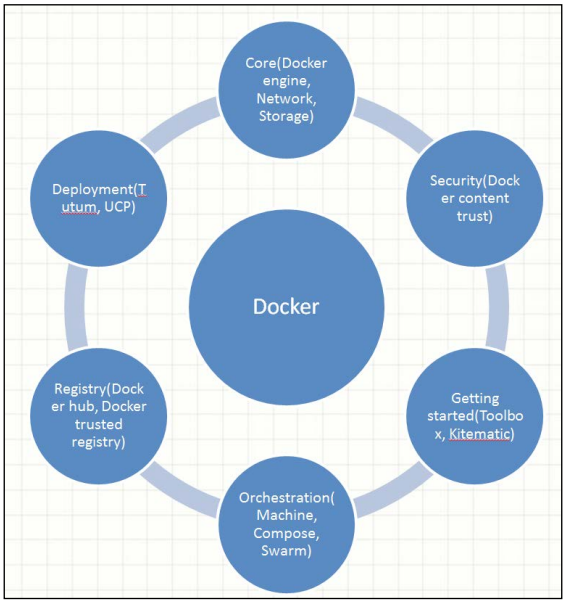
- Clair analyzes each Container layer by querying the package manager-related files and compares them against the vulnerability database available in the particular Linux distribution to check whether the particular Container layer is vulnerable
- Clair makes an index-directed graph of each Container layer, and this speeds up the analysis of a lot of Container images sharing layers
- Clair currently supports CentOS, Ubuntu, and Debian Linux distributions

Clair is still in its early stages and under active development.

The Docker roadmap

Docker has transitioned from providing Container runtime to a Container platform. Docker provides both open source solutions as well as commercial products around Containers.

The following diagram shows different Docker products around Core Docker, Security, Orchestration, Registry, and Deployment as of November 2015:



The following are some new projects announced recently by Docker.

Tutum

Tutum makes it easy to build, deploy, and manage Containerized applications and is available as a SaaS application. An application can be a single- or multi-container application. Tutum integrates well with the Docker hub.

UCP

UCP is Docker's commercial offering to provide on-premise Container deployment solutions. UCP integrates with the Docker trusted registry as well as with enterprise services such as LDAP and **Role-based access control** (**RBAC**). UCP also integrates with all other Docker services such as Networking, Compose, and Swarm. UCP is in the beta phase currently.

Nautilus

This project is targeted towards Container vulnerability detection. This is similar to the Clair project from CoreOS. Nautilus is still in the very early stages.

Microservices infrastructure

In this section, we will cover an overview of microservice infrastructure components and examples of a few solution providers.

Platform choices

The following are some design decisions/platform choices that customers who are developing and deploying microservices need to make. The following examples are just a sample set and do not cover all the providers.

IaaS vs PaaS: This choice applies for local data centers as well as for Cloud providers. In the earlier section, we covered the comparison between Container and PaaS models. Here, the trade-offs are flexibility versus time-to-market.

Local data center versus cloud providers: This is mostly a cost versus time trade-off.

Base OS: The choice here is either going with Container-optimized OSes such as CoreOS, Rancher, or Atomic or traditional OSes such as Ubuntu or Fedora. For pure microservice architecture, Container-optimized OSes are definitely worth pursuing.

VM Orchestration: VMs and Containers have different use cases and will continue to live together. There will be scenarios where VMs will be used standalone or Containers will run on top of VMs. There are open source solutions such as OpenStack and commercial solutions from VMWare for VM Orchestration.

Container runtime: Choices here are Docker, Rkt, or LXC.

Networking: Container orchestration systems such as Kubernetes typically integrate networking. As networking support is provided as plugins, it can be swapped with a different implementation if necessary. Some examples of networking plugins include Weave, Calico, and Contiv.

Storage: We need to evaluate between dedicated storage versus stateful Containers. Choices for stateful Containers are GlusterFS, Ceph, or Flocker.

Container Orchestration: Choices here are Kubernetes, Docker Swarm, Mesos, and so on.

Service discovery and DNS: This can be built manually using building blocks mentioned in previous sections, or if we choose a container orchestration system such as Kubernetes, it's already integrated with this.

CI and CD: This can be manually built or we can use packaged solutions from Codeship, CircleCI, or Shippable.

Monitoring and logging system: Examples are Sysdig or Logentries. We covered more details on Monitoring and logging in *Chapter 10, CoreOS and Containers – Troubleshooting and Debugging*.

Solution providers

As we have seen throughout this book, there are many hardware and software components that comprise the infrastructure to create and deploy microservices. We can think of each component as a LEGO block and there are numerous ways of bringing these LEGO blocks together. Customers have the following three choices:

- Integrating all the infrastructure components by themselves
- Going with solution providers who integrate these components and give an opinionated architecture
- Choosing a hybrid solution between the previous two options, where we can choose reference architecture and replace a few components based on specific needs

The following are some commercial and open source integrated solutions available. The list is not extensive and some of these solutions do not integrate all the components:

- Tectonic Enterprise from CoreOS.
- Google Container service
- AWS Container service
- Cisco's Mantl project
- OpenStack Magnum

Summary

In this chapter, we covered some of the production considerations in deploying microservice-based distributed infrastructure, and this includes CoreOS, Docker, and the associated ecosystem. Cloud companies such as Google, Amazon, and Facebook have used microservices and Container-based technologies for quite a long time and they have learned the best practices and pitfalls based on their experience.

The issue till now has been the replication of approaches and not having a common standard/approach. The trend in the last few years has been that these companies as well as many start-ups such as CoreOS and Docker are willing to develop technologies and work together in an open manner that helps the entire industry. A big contributor to this is open source software development, and many big companies are willing to develop software in the open now. Obviously, commercial solutions around open source technologies will continue to thrive as the industry still needs to make money to survive.

Container technology and microservices are the biggest trends in the software industry currently. Customers have many options and this includes both open source and commercial solutions. At this point, there is a need to put together different technologies/products to create a complete solution for microservices infrastructure. As these technologies mature, integrated open solutions with a pluggable architecture will win over the long term.

References

- Registrator: http://gliderlabs.com/registrator/latest/user/quickstart/
- Ansible reference: https://docs.ansible.com/
- Managing CoreOS with Ansible: https://coreos.com/blog/managing-coreos-with-ansible/ and https://github.com/defunctzombie/ansible-coreos-bootstrap
- Ansible Docker module: http://docs.ansible.com/ansible/docker_module.html
- CoreOS and Docker: https://developer.rackspace.com/blog/ansible-and-docker/ and http://opensolitude.com/2015/05/26/building-docker-images-with-ansible.html
- CI pipeline with Docker: https://www.docker.com/sites/default/files/UseCase/RA_CI%20with%20Docker_08.25.2015.pdf
- Containers and PaaS: http://cloudtweaks.com/2014/12/paas-vs-docker-heated-debate/ and http://thenewstack.io/docker-is-driving-a-new-breed-of-paas/
- Container security with SELinux and CoreOS: https://coreos.com/blog/container-security-selinux-coreos/
- CoreOS Ignition: https://github.com/coreos/ignition and https://coreos.com/ignition/docs/latest/examples.html
- CoreOS DEX: https://github.com/coreos/dex, https://coreos.com/blog/announcing-dex/, and https://www.youtube.com/watch?v=QZgkJQiI_gE
- Clair for Container vulnerability analysis: https://coreos.com/blog/vulnerability-analysis-for-containers/ and https://github.com/coreos/clair
- Docker Tutum and UCP: https://blog.docker.com/2015/11/dockercon-eu-2015-docker-universal-control-plane/, https://www.docker.com/tutum, and https://www.docker.com/universal-control-plane
- Mantl project: https://github.com/CiscoCloud/microservices-infrastructure and http://mantl.io/

Further reading and tutorials

- Service discovery: `http://progrium.com/blog/2014/07/29/understanding-modern-service-discovery-with-docker/`
- Ansible with Docker on Rancher: `http://rancher.com/using-ansible-with-docker-to-deploy-a-wordpress-service-on-rancher/`
- Stateful Containers: `http://techcrunch.com/2015/11/21/i-want-to-run-stateful-containers-too/`
- Codeship, Shippable, and CircleCI: `https://scotch.io/tutorials/speed-up-your-deployment-workflow-with-codeship-and-parallelci`, `https://circleci.com/docs/docker`, `https://blog.codeship.com/continuous-integration-and-delivery-with-docker/`, and `http://docs.shippable.com/`
- Comparing CI/CD solutions: `https://www.quora.com/What-is-the-difference-between-Bamboo-CircleCI-CIsimple-Ship-io-Codeship-Jenkins-Hudson-Semaphoreapp-Shippable-Solano-CI-TravisCI-and-Wercker`
- Containers and PaaS: `https://labs.ctl.io/flynn-vs-deis-the-tale-of-two-docker-micro-paas-technologies/` and `https://www.youtube.com/watch?v=YydhEEgOoDg`
- Ignition presentation: `https://www.youtube.com/watch?v=ly3uwn0HzBI`
- Jenkins Docker plugin: `https://wiki.jenkins-ci.org/display/JENKINS/Docker+Plugin`
- Continuous delivery with Docker and Jenkins: `https://www.docker.com/sites/default/files/UseCase/RA_CI%20with%20Docker_08.25.2015.pdf` and `https://pages.cloudbees.com/rs/083-PKZ-512/images/Docker-Jenkins-Continuous-Delivery.pdf`

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

T

U

V

W