Mastering CoreOS

CoreOS makes Google and Amazon-style Cloud infrastructure available for anyone building their own private Cloud. This book covers the CoreOS internals and the technologies used in the deployment of container-based distributed applications. It starts with an overview of CoreOS and distributed application development while sharing knowledge on related tools and technologies. Critical CoreOS services and networking and storage considerations for CoreOS are covered next.

In latter half of the book, you will learn about Container runtime systems such as Docker and Rkt and Container Orchestration using Kubernetes. You will also find out about the integration of popular orchestration solutions such as OpenStack, the AWS Container service, and the Google Container Engine with CoreOS and Docker. Lastly, we cover troubleshooting as well as production considerations.

Who this book is written for

This book is intended for Cloud application developers and Cloud infrastructure administrators. If you are looking to deploy a CoreOS cluster or you already have a CoreOS cluster that you want to manage better in terms of performance, security, and scaling, then this book is perfect for you. To follow the hands-on stuff, you need to have a Google and an AWS Cloud account and be able to run CoreOS VMs on your machine. A basic understanding of public and private clouds, Containers, Docker, Linux, and CoreOS is required.

What you will learn from this book

 Install CoreOS on a VM, on the Cloud, and find out how to keep your cluster secure and up to date
 Configure and troubleshoot key CoreOS services, such as etcd, systemd, and fleet, for distributed application deployment
 Study container networking using CoreOS Flannel and other solutions, such as Docker libnetwork, Weave, and Calico
 Explore the container file system and container volume management using Docker volume, NFS, GlusterFS, and Flocker
 Get to know the internals of container technologies such as Docker, Rkt, and Container orchestration using Openstack, Kubernetes and Docker native solutions
 Troubleshoot CoreOS cluster and Containers using monitoring and logging tools and master production techniques such as staging, security, and automation

Sreenivas Makam

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

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'CoreOS Overview'
- A synopsis of the book’s content
- More information on Mastering CoreOS
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.
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 – Etcld, Systemd, and Fleet, discusses the internals of CoreOS critical services – Etcld, 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.
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.

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

<table>
<thead>
<tr>
<th>Feature</th>
<th>CoreOS</th>
<th>Rancher OS</th>
<th>Atomic</th>
<th>Ubuntu snappy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Registry</td>
<td>Docker hub and Quay</td>
<td>Docker hub</td>
<td>Docker hub</td>
<td>Docker hub</td>
</tr>
<tr>
<td>Debugging</td>
<td>Toolbox</td>
<td>Rancher’s own tools and external tools</td>
<td>RedHat tools</td>
<td>Ubuntu debug tools</td>
</tr>
<tr>
<td>Security</td>
<td>SELinux can be turned on</td>
<td>There is a plan to add SELinux and AppArmor support</td>
<td>SELinux enabled by default, additional security</td>
<td>AppArmor security profile can be used</td>
</tr>
</tbody>
</table>
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 Makan <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 Overview

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:

```
$ uname -a
Linux core-01 4.1.7-coreos-1 #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:

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

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

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

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:

```bash
[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
TimeoutStopSec=15
LimitNOFILE=1048576
execStart=/usr/lib/systemd/system/docker.sock start --host=unix:///var/run/docker.sock

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

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

```bash
[Unit]
Description=Docker Socket for the API
PartOf=docker.service

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

[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 book

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
ExecStart=/usr/bin/docker run --name hello1 busybox /bin/sh -c "while true; do echo Hello World; sleep 1; done"
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:

```bash
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`:

```
Hello1 Service - My Service
Loaded: loaded (/etc/systemd/system/hello1.service; enabled; vendor preset: disabled)
Active: active (running) since Tue 2016-08-02 13:38:44 UTC; 13mins ago
Process: 2976 ExecStart=/usr/bin/docker run --name hello1 busybox /bin/sh -c while true; do echo Hello World; sleep 1; done
Main PID: 2976 (docker)
Memory: 9.9M
CPU: 52ms
```

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:

```
  NAME   COMMAND       CREATED             STATUS       PORTS               LABELS
hello1  /bin/sh      4 minutes ago      Up 4 minutes
```

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:

```bash
$ docker stop hello1
```

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

```
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: 2861 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:

```
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
```

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:

```
[15:42:01] core-01: systemd[1]: hello1.service: Main process exited, code=exited, status=137
Sep 08 15:42:01 core-01 systemd[2891]: hello1
Sep 08 15:42:33 core-01 systemd[1]: hello1.service: Service hold-off time over, scheduling restart.
Sep 08 15:42:33 core-01 systemd[1]: Starting My Service...
Sep 08 15:42:33 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:

```bash
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.

<table>
<thead>
<tr>
<th>Cluster size</th>
<th>Majority</th>
<th>Failure tolerance</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>2</td>
</tr>
</tbody>
</table>
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
1062024b6874f64: name=m2le50tase80hbc396d20d853d4d7e peerURL=http://172.17.0.103:2380 clientURL=http://172.17.0.103:2379
76921df4dfe807d1: name=5ed324e0553af9d914f4419a7bce peerURL=http://172.17.0.102:2380 clientURL=http://172.17.0.102:2379
```

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
member 1062024b6874f64 is healthy
member 1413663a07f117 is healthy
member 76921df4dfe807d1 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 ~]# etcdctl stats
```

We can see that the member ID matches with the leader ID, 41419684778c117.

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

```
[core@core-01 ~]# etcdctl stats
```

**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.
CoreOS Overview

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:

```
MACHINE   IP            META DATA
18ec8345... 172.17.8.101 compute=web, rack=one
d9ce32f6... 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=systemd.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:

<table>
<thead>
<tr>
<th>UNIT</th>
<th>MACHINE</th>
<th>ACTIVE</th>
<th>SUB</th>
</tr>
</thead>
<tbody>
<tr>
<td>helloglobal.service</td>
<td>18ce8345.../172.17.8.101</td>
<td>active</td>
<td>running</td>
</tr>
<tr>
<td>helloglobal.service</td>
<td>d9ec82f6.../172.17.8.102</td>
<td>activating</td>
<td>start-pre</td>
</tr>
<tr>
<td>helloglobal.service</td>
<td>ea28e504.../172.17.8.103</td>
<td>activating</td>
<td>start-pre</td>
</tr>
</tbody>
</table>

### 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 identify 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:

```markdown
[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 13ce8345.../172.17.8.101
core@core-01 ~ $ fleetctl start redisdb.service
Unit redisdb.service launched on ea28e504.../172.17.8.103
```

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:
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**:

![Flannel diagram](image-url)
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:

```json
{
    "Network": "10.0.0.0/8",
    "SubnetLen": 28,
    "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:

![Data Path Diagram]

**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:

```bash
[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.
CoreOS Overview

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

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:

```
<table>
<thead>
<tr>
<th>Container Specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>APPC</td>
</tr>
<tr>
<td>OCI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Container Runtime</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rkt, Kurma, Jetpack</td>
</tr>
<tr>
<td>RUNC</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Container platform</th>
</tr>
</thead>
<tbody>
<tr>
<td>CoreOS</td>
</tr>
<tr>
<td>Docker</td>
</tr>
</tbody>
</table>
```

**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=gVGZhRjvo0
• 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
Where to buy this book

You can buy Mastering CoreOS from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.