Learning CoreOS

CoreOS is an open-source operating system developed upon the Linux kernel. It is a minimal operating system layer and takes a different approach to automating the deployment of containers. CoreOS helps users immensely to create systems that are easy to scale and manage, making life easier for all.

This book will teach you what CoreOS is and its benefits as a cloud orchestration platform. First, we’ll set up a simple CoreOS instance with a single node in the cluster and run a Docker container inside the CoreOS instance. Next, you’ll be introduced to Fleet and systemd, and will deploy and distribute Docker services using Fleet. Later, you’ll be briefed about running services in a cluster with constraints and publishing the services already running on the cluster. We conclude by teaching you about advanced container networking. By the end of the book, you will know the salient features of CoreOS and will be able to deploy, administrate, and secure a CoreOS environment.

Who this book is written for
This book is for application developers who would like to gain knowledge about CoreOS in order to deploy a cloud application on a cluster of cloud servers. You do not need to have any knowledge of CoreOS.

What you will learn from this book
- Understand the benefits of CoreOS as a cloud orchestration platform
- Learn about lightweight containers and various container frameworks, such as Docker and RKT, in CoreOS
- Manage services and containers across the cluster using Fleet and systemd
- Set up a CoreOS environment using Vagrant
- Create and manage CoreOS clusters
- Discover the service parameters using etcd
- Find out about chaining services running on the cluster using Flannel / Rudder and Weave
- Create a virtual tenant network and service chaining using OVS

Who this book is written for
This book is for application developers who would like to gain knowledge about CoreOS in order to deploy a cloud application on a cluster of cloud servers. You do not need to have any knowledge of CoreOS.

What you will learn from this book
- Understand the benefits of CoreOS as a cloud orchestration platform
- Learn about lightweight containers and various container frameworks, such as Docker and RKT, in CoreOS
- Manage services and containers across the cluster using Fleet and systemd
- Set up a CoreOS environment using Vagrant
- Create and manage CoreOS clusters
- Discover the service parameters using etcd
- Find out about chaining services running on the cluster using Flannel / Rudder and Weave
- Create a virtual tenant network and service chaining using OVS
In this package, you will find:

- The author's biography
- A preview chapter from the book, Chapter 1 'CoreOS, Yet Another Linux Distro?'
- A synopsis of the book’s content
- More information on Learning CoreOS
About the Authors

Kingston Smiler. S is a seasoned professional with 12 years of experience in software development and presales, encompassing wide range of skill set, roles, and industry verticals. He has solid expertise in data communication networking and software-based switching and routing solutions, and virtualization platforms such as OpenStack, OpenDaylight controller, Docker Containers, and CoreOS. He is currently working as an advisor and technical consultant for networking companies in the development of Layer2 and Layer3 IP protocols.

He also has working experience in building IoT sensor networks with IoT OSes such as RIOT and Contiki; SoCs such as Arduino, Raspberry Pi, and Intel Galileo; and IoT protocols such as 802.14.5 (Zigbee), 6lowpan, RPL, CoAP, and MQTT. He is interested in building small-scale robots using Arduino and ROS Robotics platforms.

Active in various networking standard bodies such as IETF, IEEE, and ONF, Kingston has proposed two drafts in TRILL WG and one draft in MPLS WG of IETF. With the current surge in SDN, virtualization, and NFV, his primary focus is towards these areas. He completed a bachelor of engineering degree in computer science from Madras University.

Kingston is also the author of *OpenFlow Cookbook, Packt Publishing*. 
Shantanu Agrawal has over 15 years of experience in the telecom industry working in systems having high transaction rates with scalable architectures. He has extensive experience of consulting for solutions, designing, and implementing high-performing and highly available software. He has exposure to the complete life cycle of the product development and deployment challenges. During his journey, he has worked on different hardware platforms and operating systems, such as proprietary UNIX-based hardware, embedded systems with real-time operating systems, Linux, Solaris, and so on. He has experience in the core network elements and protocols used in GSM/UMTS/LTE networks. He graduated from IIT BTU and did post-graduation from BITS Pilani.
As more and more applications are moving towards the cloud with server virtualization, there is a clear necessity to deploy the user applications and services very fast and make them reliable with assured SLA by deploying the services in a right set of servers. This becomes more complex when these services are dynamic in nature, which results in making these services autoprovisioned and autoscaled over a set of nodes. The orchestration of the user application is not limited to deploy the services in the right set of server or virtual machines rather to be extended to provide network connectivity across these services to provide **Infrastructure as a Service (IaaS)**. Compute, network, and storage are the three main resources to be managed by the cloud provider in order to provide (IaaS). Currently, there are various mechanisms to handle these requirements in more abstract fashion. There are multiple cloud orchestration frameworks, which can manage the compute, storage, and networking resources. OpenStack, Cloud Stack, and VMware vSphere are some of the cloud platforms that orchestrate these resource pools and provide IaaS.

The server virtualization provided by a Virtual Machine (VM) has its own overhead of running a separate instance of the operating system on every virtual machine. This brings down the number of VM instances that can be run on the server, which heavily impacts the operational expense. As Linux namespace, containers technologies, such as docker and rkt, are gaining its popularity; one more level of server virtualization can be introduced by deploying the application services inside a container rather than VM However, there is a necessity of an orchestration and clustering framework for the containers or dockers for deploying the services in the cluster, the discovery of the service and service parameters, providing network across these containers or dockers, and so on. CoreOS is developed for this purpose.

CoreOS is a light-weight cloud service orchestration operating system based on Google Chrome. CoreOS is developed primarily for orchestrating applications/services over a cluster of nodes. CoreOS extends the existing services provided by Linux to work for a distributed cluster and not limited to a single node.
What this book covers

Chapter 1, CoreOS, Yet Another Linux Distro?, explains the basics of containers, dockers, and high-level architecture of CoreOS.

Chapter 2, Setting Up Your CoreOS Environment, teaches you how to set up and run CoreOS with a single machine using Vagrant and VirtualBox. It also covers how to create and run docker images and get familiarized with the important configuration files and their contents.

Chapter 3, Creating Your CoreOS Cluster and Managing the Cluster, teaches you how to set up the CoreOS cluster with multiple machines. You will also learn how machines are discovered and services are scheduled on those machines. Also, you will learn about starting and stopping a service using Fleet.

Chapter 4, Managing Services with User-Defined Constraints, gives an introduction about service constraints, which helps to deploy services on suitable members.

Chapter 5, Discovering Services Running in Cluster, explains the need and mechanism for the discovery of services running on a cluster. Also, you will learn about two important tools, which are used widely for service discovery: etcdctl and curl.

Chapter 6, Service Chaining and Networking Across Services, explains the importance of the container communications and the various possibilities provided by CoreOS and docker to provide the communication.

Chapter 7, Creating a Virtual Tenant Network and Service Chaining Using OVS, explains the importance of OVS in container communications and the various advantages provided by OVS. The chapter details how the services deployed by different customers/tenants across the CoreOS cluster can be linked/connected using OVS.

Chapter 8, What Next?, touches upon some advanced Docker and Core OS topics and also discusses about what is coming up in CoreOS.
CoreOS, Yet Another Linux Distro?

As more and more applications move toward the cloud with server virtualization, there is a clear necessity for deploying user applications and services very fast and reliably with assured SLA by deploying the services in the right set of servers. This becomes more complex when these services are dynamic in nature, which results in making these services auto-provisioned and auto-scaled over a set of nodes. The orchestration of the user application is not limited to deploying the services in the right set of servers or virtual machines, rather to be extended to provide network connectivity across these services to provide Infrastructure as a Service (IaaS). Compute, network, and storage are the three main resources to be managed by the cloud provider in order to provide IaaS. Currently, there are various mechanisms to handle these requirements in a more abstract fashion. There are multiple cloud orchestration frameworks that can manage compute, storage, and networking resources. OpenStack, Cloud Stack, and VMware vSphere are some of the cloud platforms that perform orchestration of these resource pools and provide IaaS. For example, the Nova service in OpenStack manages the compute resource pool and creates VMs; the Neutron service provides the required information to provide virtual network connectivity across VMs; and so on.

The IaaS cloud providers should provide all three resources on-demand to the customers, which provide a pay-as-you-go model. The cloud provider maintains these resources as a pool and allocates the resource to a customer on-demand. This provides flexibility for the customer to start and stop the services based on their business needs and can save their OPEX. Typically, in an IaaS model, the cloud service provider offers these resources as a virtualized resource, that is, a virtual machine for compute, a virtual network for network, and virtual storage for storage. The hypervisor running in the physical server/compute nodes provides the required virtualization.
Typically, when an end user requests an IaaS offering with a specific OS, the cloud provider creates a new VM (Virtual Machine) with the OS requested by the user in their cloud server infrastructure. The end user can install their application in this VM. When the user requests more than one VM, the cloud provider should also provide the necessary network connectivity across these VMs in order to provide connectivity across the services running inside these VMs. The cloud orchestration framework takes care of instantiating the VMs in one of the available compute nodes in the cluster, along with associated services like providing virtual network connectivity across these VMs. Once the VM has been spawned, configuration management tools like Chef or Puppet can be used to deploy the application services over these VMs. Theoretically, this works very well.

There are three main problems with this approach:

- All the VMs in the system should run their own copy of the operating system with their own memory management and virtual device drivers. Any application or services deployed over these VMs will be managed by the OS running in the VM. When there are multiple VMs running in a server, all the VMs run a separate copy of OS, which results in overhead with respect to CPU and memory. Also, as the VMs run their own operating system, the time taken to boot/bring up a VM is very high.

- The operating system doesn't provide service-level virtualization that is running a service/application over a set of VMs which are part of cluster. The OS running in the VM is a general purpose operating system that lacks the concept of clustering and deploying the application or service over this cluster. In short, the operating system provides machine-level virtualization and not service-level virtualization.

- The management effort required to deploy a service/software from a development to a production environment is very high. This is because each software package typically has dependencies with other software. There are thousands of packages; each package comes with a different set of configuration, with most combinations of configurations having dependency with respect to performance and scaling.

CoreOS addresses all these problems. Before looking into how CoreOS solves these problems, we will look at a small introduction to CoreOS.
Introduction to CoreOS

CoreOS is a lightweight cloud service orchestration operating system based on Google's Chrome OS. CoreOS is developed primarily for orchestrating applications/services over a cluster of nodes. Every node in the cluster runs CoreOS and one of the CoreOS nodes in the cluster will be elected as the master node by the etcd service. All the nodes in the cluster should have connectivity to the master node. All the slave nodes in the system provide information about the list of services running inside their system, along with the configuration parameter to the master node. In order to do this, we may have to configure fleet units in such a way that when we start a fleet unit with the `fleetctl` command, it should push its details such as IP and port to the etcd service. It is the responsibility of the master node to receive the service information and publish to all the other nodes in the cluster. In normal circumstances, the slave nodes won't talk to each other regarding service availability. The etcd service running in all the nodes in the cluster is responsible for electing the master node. All nodes in the system interact with the etcd service of the master node to get the service and configuration information of the services running in all other nodes. The following diagram depicts the CoreOS cluster architecture, wherein all the nodes in the cluster run CoreOS and other vital components of CoreOS like etcd, systemd, and so on. The etcd and fleet services are used for service discovery and cluster management respectively. In this, all three nodes are configured with the same cluster ID, so that all these nodes can be part of a single cluster. It is not possible for a node to be part of multiple clusters.
CoreOS, Yet Another Linux Distro?

All the applications or services are deployed as a Linux container in the CoreOS. The Linux container provides a lightweight server virtualization infrastructure without running its own operating system or any hypervisor. It uses the operating system-level virtualization techniques provided by the host OS using the namespace concept. This provides drastic improvements in terms of scaling and performance of virtualization instances running over the physical server. This addresses the first issue of running the application inside a VM.

The following diagram depicts the difference between applications running inside a VM and applications running in an LXC container. In the following diagram, the VM way of virtualization has a guest OS installed in the VM along with the host OS. In a Linux container-based implementation, the container doesn't have a separate copy of the operating system; rather, it uses the service provided by the host operating system for all the OS-related functionalities.

CoreOS extends the existing services provided by Linux to work for a distributed cluster and not limited to a single node. As an example, CoreOS extends the system management service provided by most of the Linux distribution for starting, stopping, or restarting any applications/services to run on a cluster of nodes rather than a single node using the fleet tool. Instead of running an application limited to its own node, the services are submitted to fleet, which acts as a cluster manager and instantiates the service in any one of the nodes in the cluster. It is also possible to launch the container in a specific set of nodes by applying a constraint. This addresses the second issue with using VMs, discussed earlier in this chapter.
CoreOS uses Docker/Rocket as a container to deploy services inside the CoreOS cluster. Docker provides an easy way of bundling a service and its dependent module as a single monolithic image that can be shipped from development. In the deployment, the DevOps person can simply fetch the docker container from the development person and can deploy directly into the CoreOS nodes without performing any operations like building a compilation or build environment and rebuilding the image on the target platform and so on. This bridges the gap between the development and deployment of a service. This addresses the third issue with using VM, discussed earlier in this chapter.

**CoreOS versus other Linux distributions**

Even though CoreOS is yet another Linux distribution like Fedora/Centos, the key difference between CoreOS and other standard Linux distributions are as follows:

- CoreOS is not designed to run any applications or services directly. Any application to be run inside CoreOS should be deployed as a container (which can either be Docker/Rocket). So it is not possible to install any software packages in CoreOS and hence CoreOS doesn't have any installation software packages like `yum`, `apt`, and so on. In short, CoreOS is a stripped-down version of a Linux distribution that doesn't have any inbuilt user applications or library installed.

- Most of the Linux distributions are meant to run as a host operating system either in a data center server or in a typical desktop PC. They are not developed to manage a cluster of nodes/the cloud; rather, they will be part of the cloud that is being managed by other cloud orchestration platforms. However, CoreOS is a Linux distribution that is builtout for the management of a massive server infrastructure with clustering. The CoreOS cluster is a group of physical or virtual machines that runs CoreOS with the same cluster ID. The services running in the cluster nodes are managed by fleet, which is the CoreOS orchestration tool. Software updates in a traditional Linux distribution are done by updating the packages one by one. However, CoreOS supports a scheme called **fast patch**, wherein the entire CoreOS OS is updated once. The **CoreUpdate** program is used for updating CoreOS in a server, cluster, or complete data center.

- CoreOS is extremely lightweight when compared to traditional Linux distributions.
CoreOS, Yet Another Linux Distro?

CoreOS high-level architecture

The CoreOS node in a cluster comprises the following main components:

- etcd
- systemd
- fleet
- Docker/Rocket containers

The CoreOS node runs etcd, systemd, and the fleet service in all of the nodes in the cluster. etcd, which is running in all the nodes, talk to each other and elects one node as the master node. All the services running inside the node will be advertised to this master node, which makes etcd provide a service discovery mechanism. Similarly, fleetd running in different nodes maintains the list of services running in different nodes in its service pool, which provides service-level orchestration. fleetctl and etcdctl are command-line utilities to configure the fleet and etcd utilities respectively.
Refer to subsequent sections of this chapter to understand the functionality of each component in detail.

These components together provide three main functionalities for CoreOS as follows:

- Service discovery
- Cluster management
- Container management

**Service discovery**

In the CoreOS environment, all user applications are deployed as services inside a container that can either be a Docker container or a Rocket container. As different applications/services are running as separate containers in the CoreOS cluster, it is inevitable to announce the services provided by each node to all the nodes in the cluster. Along with service availability, it is also required that each service advertises the configuration parameters to other services. This service advertisement is very important when the services are tightly coupled and dependent on each other. For example, the web service should know details about the database services, about the connection string, or type of database and so on. CoreOS provides a way for each service to advertise its service and configuration information using the etcd service. The data announced to the etcd service will be given/announced to all the nodes in the cluster by the master node.

**etcd**

*etcd* is a distributed key value store that stores data across the CoreOS cluster. The etcd service is used for publishing services running on a node to all the other nodes in the cluster, so that all the services inside the cluster discover other services and configuration details of other services. etcd is responsible for electing the master node among the set of nodes in the cluster. All nodes in the cluster publish their services and configuration information to the etcd service of the master node, which provides this information to all the other nodes in the cluster.
Container management

The key element of the CoreOS building block is a container that can either be Docker or Rocket. The initial version of CoreOS officially supports Docker as the means for running any service application in the CoreOS cluster. In the recent version, CoreOS supports a new container mechanism called Rocket, even though CoreOS maintains backward compatibility with Docker support. All customer applications/services will be deployed as a container in the CoreOS cluster. When multiple services are running inside a server for different customers, it is inevitable to isolate the execution environment from one customer to another customer. Typically, in a VM-based environment, each customer will be given a VM and inside this VM the customer can run their own service, which provides complete isolation of the execution environment between customers. The container also provides a lightweight virtualization environment without running a separate copy of the VM.

Linux Container

Linux Container (LXC) is a lightweight virtualization environment provided by the Linux kernel to provide system-level virtualization without running a hypervisor. LXC provides multiple virtualized environments, each of them being inaccessible and invisible from the other. Thus, an application that is running inside one Linux container will not have access to the other containers.

LXC combines three main concepts for resource isolation as follows:

- Cgroups
- Namespaces
- Chroot
The following diagram explains in detail about LXC and the utilities required to provide LXC support:

Libvirt is a 'C' library toolkit that is used to interact with the virtualization capabilities provided by the Linux kernel. It acts as a wrapper layer for accessing the APIs exposed by the virtualization layer of the kernel.
cgroups

Linux **cgroups** is a feature provided by the kernel to restrict access to system resource for a process or set of processes. The Linux cgroup provides a way to reserve or allocate resources, such as CPU, system memory, network bandwidth and so on, to a group of processes/tasks. The administrator can create a cgroup and set the access levels for these resources and bind one or more processes to these groups. This provides fine-grained control over the resources in the system to different processes. This is explained in detail in the next diagram. The resources mentioned on the left-hand side are grouped into two different cgroups called cgroups-1 and cgroups-2. task1 and task2 are assigned to cgroups-1, which makes only the resources allocated for cgroups-1 available for task1 and task2.

Managing cgroups consists of the following steps:

1. Creation of cgroups.
2. Assign resource limit to the cgroup based on the problem statement. For example, if the administrator wants to restrict an application not to consume more that 50 percent of CPU, then he can set the limit accordingly.
3. Add the process into the group.
As the creation of cgroups and allocation of resources happens outside of the application context, the application that is part of the cgroup will not be aware of cgroups and the level of resource allocated to that cgroup.

**Namespace**

namespace is a new feature introduced from Linux kernel version 2.6.23 to provide resource abstraction for a set of processes. A process in a namespace will have visibility only to the resources and processes that are part of that namespace alone. There are six different types of namespace abstraction supported in Linux as follows:

- PID/Process Namespace
- Network Namespace
- Mount Namespace
- IPC Namespace
- User Namespace
- UTS Namespace

**Process Namespace** provides a way of isolating the process from one execution environment to another execution environment. The processes that are part of one namespace won't have visibility to the processes that are part of other namespaces. Typically, in a Linux OS, all the processes are maintained in a tree with a child-parent relationship. The root of this process tree starts with a specialized process called init process whose process-id is 1. The init process is the first process to be created in the system and all the process that are created subsequently will be part of the child nodes of the process tree. The process namespace introduces multiple process trees, one for each namespace, which provides complete isolation of the processes running across different namespaces. This also brings the concept of a single process to have two different pids: one is the global context and other in the namespace context. This is explained in detail in the following diagram.
In the following diagram, for namespace, all processes have two process IDs: one in the namespace context and the other in the global process tree.

**Network Namespace** provides isolation of the networking stack provided by the operating system for each container. Isolating the network stack for each namespace provides a way to run multiple same services, say a web server for different customers or a container. In the next diagram, the physical interface that is connected to the hypervisor is the actual physical interface present in the system. Each container will be provided with a virtual interface that is connected to the hypervisor bridging process. This hypervisor bridging process provides inter-container connectivity across the container, which provides a way for an application running in one container to talk to another application running in another container.
Chroot

Chroot is an operation supported by Linux OS to change the root directory of the current running process, which apparently changes the root directory of its child. The application that changes the root directory will not have access to the root directory of other applications. Chroot is also called chroot jail.

Combining the cgroups, namespace, and chroot features of the Linux kernel provides a sophisticated virtualized resource isolation framework with clear segregation of the data and resources across various processes in the system.
In LXC, the chroot utility is used to separate the filesystem, and each filesystem will be assigned to a container that provides each container with its own root filesystem. Each process in a container will be assigned to the same cgroup with each cgroup having its own resources providing resource isolation for a container.

**Docker**

Docker provides a portable way to deploy a service in any Linux distribution by creating a single object that contains the service. Along with the service, all the dependent services can be bundled together and can be deployed in any Linux-based servers or virtual machine.

Docker is similar to LXC in most aspects. Similar to LXC, Docker is a lightweight server virtualization infrastructure that runs an application process in isolation, with resource isolation, such as CPU, memory, block I/O, network, and so on. But along with isolation, Docker provides "Build, Ship and Run" modeling, wherein any application and its dependencies can be built, shipped, and run as a separate virtualized process running in a namespace isolation provided by the Linux operating system.

Dockers can be integrated with any of the following cloud platforms: Amazon Web Services, Google Cloud Platform, IBM Bluemix, Jelastic, Jenkins, Microsoft Azure, OpenStack Nova, OpenSVC, and configuration tools such as Ansible, CFEngine, Chef, Puppet, Salt, and Vagrant. The following are the main features provided by Docker.

The main objective of Docker is to support micro-service architecture. In micro-service architecture, a monolithic application will be divided into multiple small services or applications (called micro-services), which can be deployed independently on a separate host. Each micro-service should be designed to perform specific business logic. There should be a clear boundary between the micro-services in terms of operations, but each micro-service may need to expose APIs to different micro-services similar to the service discovery mechanism described earlier. The main advantage of micro-service is quick development and deployment, ease of debugging, and parallelism in the development for different components in the system. One of the main advantage of micro-services is based on the complexity, bottleneck, processing capability, and scalability requirement; every micro-service can be individually scaled.
Docker versus LXC

Docker is designed for deploying applications, whereas LXC is designed to deploy a machine. LXC containers are treated as a machine, wherein any applications can be deployed and run inside the container. Docker is designed to run a specific service or application to provide container as an application. However, when an application or service has a dependency with other services, these services can also be packed along with the same Docker image. Typically, the docker container doesn't provide all the services that will be provided by any OS, such as init systems, syslog, cron, and so on. As Docker is more focused on deploying applications, it provides tools to create a docker container and deploy the services using source code.

Docker containers are designed to have a layered architecture with each layer containing changes from the previous version. The layered architecture provides the docker to maintain the version of the complete container. Like any typical version control tools like Git/CVS, docker containers are maintained with a different version with operations like commit, rollback, version tracking, version diff, and so on. Any changes made inside the docker application will be made as a read-only layer until it is committed.

Docker-hub contains more than 14,000 containers available for various well-known services that can be downloaded and deployed very easily.

Docker provides an efficient mechanism for chaining different docker containers, which provides a good service chaining mechanism. Different docker containers can be connected to each other via different mechanisms as follows:

- Docker link
- Using docker0 bridge
- Using the docker container to use the host network stack

Each mechanism has its own benefits. Refer to Chapter 7, Creating a Virtual Tenant Network and Service Chaining Using OVS for more information about service chaining.
Docker uses libcontainer, which accesses the kernel's container calls directly rather than creating an LXC.

Rocket
Historically, the main objective of CoreOS is to run the services as a lightweight container. Docker's principle was aligning with the CoreOS service requirement with simple and composable units as container. Later on, Docker adds more and more features to make the Docker container provide more functionality than standard containers inside a monolithic binary. These functionalities include building overlay networks, tools for launching cloud servers with clustering, building images, running and uploading images, and so on. This makes Docker more like a platform rather than a simple container.

With the previously mentioned scenario, CoreOS started working on a new alternative to Docker with the following objectives:

- Security
- Composability
- Speed
- Image distribution
CoreOS announced the development of Rocket as an alternative to Docker to meet the previously mentioned requirements. Along with the development of Rocket, CoreOS also started working on an App Container Specification. The specification explains the features of the container such as image format, runtime environment, container discovery mechanism, and so on. CoreOS launched its first version of Rocket along with the App Container Specification in December 2014.

CoreOS cluster management:

**Clustering** is the concept of grouping a set of machines to a single logical system (called cluster) so that the application can be deployed in any one machine in the cluster. In CoreOS, clustering is one of the main features provided by CoreOS by running different services/docker container over the cluster of the machine. Historically, in most of the Linux distribution, services can be managed using the systemd utility. CoreOS extends the systemd service from a single node to a cluster using fleet utility. The main reason for CoreOS to choose fleet to orchestrate the services across the CoreOS cluster is as follows:

- Performance
- Journal support
- Rich syntax in deploying the services

It is also possible to have a CoreOS cluster with a combination of a physical server and virtual machines as long as all the nodes in the cluster are connected to each other and reachable. All the nodes that want to participate in the CoreOS cluster should run CoreOS with the same cluster ID.

**systemd**

systemd is an init system utility that is used to stop, start, and restart any of the Linux services or user programs. systemd has two main terminologies or concepts: unit and target. **unit** is a file that contains the configuration of the services to be started, and **target** is a grouping mechanism to group multiple services to be started at the same time.
fleet

fleet emulates all the nodes in the cluster to be part of a single init system or system service. fleet controls the systemd service at the cluster level, not in the individual node level, which allows fleet to manage services in any of the nodes in the cluster. fleet not only instantiates the service inside a cluster but also manages how the services are to be moved from one node to another when there is a node failure in the cluster. Thus, fleet guarantees that the service is running in any one of the nodes in the cluster. fleet can also take care of restricting the services to be deployed in a particular node or set of nodes in a cluster. For example, if there are ten nodes in a cluster and among the ten nodes a particular service, say a web server, is to be deployed over a set of three servers, then this restriction can be enforced when fleet instantiates a service over the cluster. These restrictions can be imposed by providing some information about how these jobs are to be distributed across the cluster. fleet has two main terminologies or concepts: engine and agents. For more information about systemd and fleet, refer to chapter Creating Your CoreOS Cluster and Managing the Cluster.

CoreOS and OpenStack

Is CoreOS yet another orchestration framework like OpenStack/CloudStack? No, it is not. CoreOS is not a standalone orchestration framework like OpenStack/CloudStack. In most server orchestration frameworks, the framework sits external to the managed cloud. But in CoreOS, the orchestration framework sits along with the existing business solution.

OpenStack is one of the most widely used cloud computing software platforms to provide IaaS. OpenStack is used for orchestrating the compute, storage, and network entities of the cloud, whereas CoreOS is used for service orchestration. Once the compute, storage, or network entities are instantiated, OpenStack doesn't have any role in instantiating services inside these VMs.

Combining the orchestration provided by OpenStack and CoreOS provides a powerful IaaS, wherein the cloud provider will have fine-grained control until the service orchestration. So CoreOS can co-exist with OpenStack, wherein OpenStack can instantiate a set of VMs that run the CoreOS instance and form a CoreOS cluster. That is, OpenStack can be used to create a CoreOS cluster as infrastructure. The CoreOS that is running inside the VM forms as a cluster and instantiates the service inside any one of the nodes in the cluster.
In the preceding diagram, OpenStack is used to manage the server farm that consists of three servers: server1, server2, and server3. When a customer is requested for a set of VMs, OpenStack creates the necessary VM in any one of these servers, as an IaaS offering. With CoreOS, all these VMs run the CoreOS image with the same cluster ID, and hence can be part of the same cluster. In the preceding diagram, there are two CoreOS clusters, each allocated for different customers. The services/applications to be run on these VMs will be instantiated by the fleet service of CoreOS, which takes care of instantiating the service in any one of the VMs in the cluster. At any point in time, OpenStack can instantiate new VMs inside the cluster in order to scale up the cluster capacity by adding new VMs running the CoreOS image with the same cluster ID, which will be a candidate for CoreOS to run new services.
Summary

CoreOS and Docker open up a new era for deploying the services in a cluster to streamline easy development and deployment of applications. CoreOS and Docker bridge the gap between the process of developing a service and deploying the service in production and make the server and service deployment less effort and less intensive work. With lightweight containers, CoreOS provides very good performance and provides an easy way to auto-scale the application with less overhead from the operator side. In this chapter, we have seen the basics of containers, Docker, and the high-level architecture of CoreOS.

In the next few chapters, we are going to see the individual building blocks of CoreOS in detail.
Where to buy this book

You can buy Learning 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.