Chapter 3.

Kubernetes Architecture - Overview

Introduction

- In this chapter, we will explore the **Kubernetes** architecture, the different components of the **master** and **worker nodes**, the cluster state management with **etcd** and the network setup requirements. We will also talk about the network specification called **Container Network Interface (CNI)**, which is used by Kubernetes.

Learning Objectives

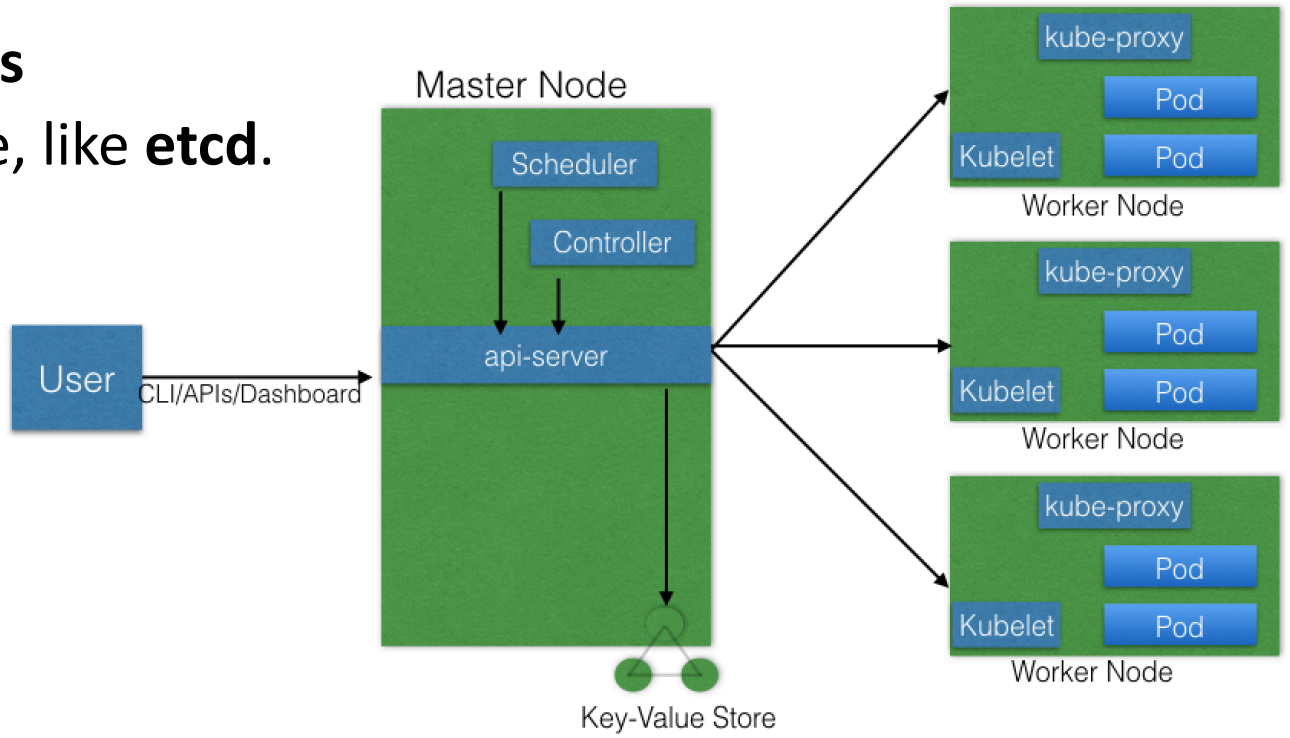
By the end of this chapter, you should be able to:

- Discuss the Kubernetes architecture.
- Explain the different components for master and worker nodes.
- Discuss about cluster state management with etcd.
- Review the Kubernetes network setup requirements.

Kubernetes Architecture

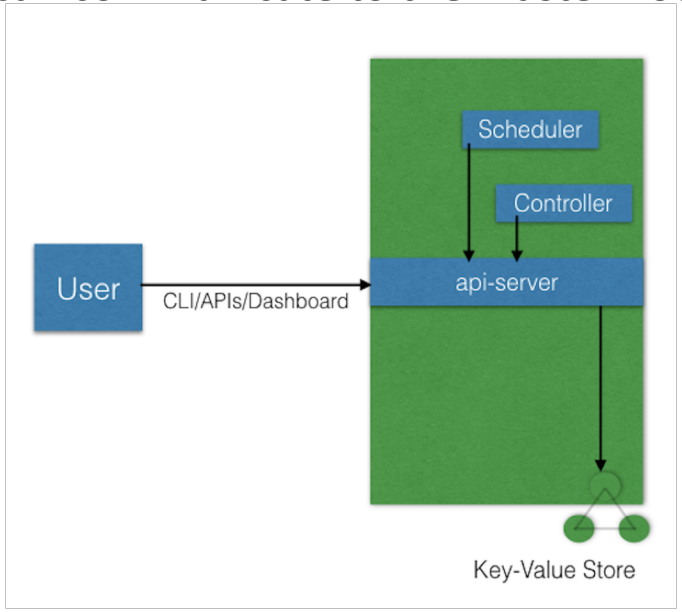
At a very high level, Kubernetes has the following main components:

- One or more **master nodes**
- One or more **worker nodes**
- Distributed key-value store, like **etcd**.



Master Node

The **master node** is responsible for **managing** the Kubernetes cluster, and it is the **entry point for all administrative tasks**. We can communicate to the master node via the **CLI, the GUI (Dashboard),** or via APIs.



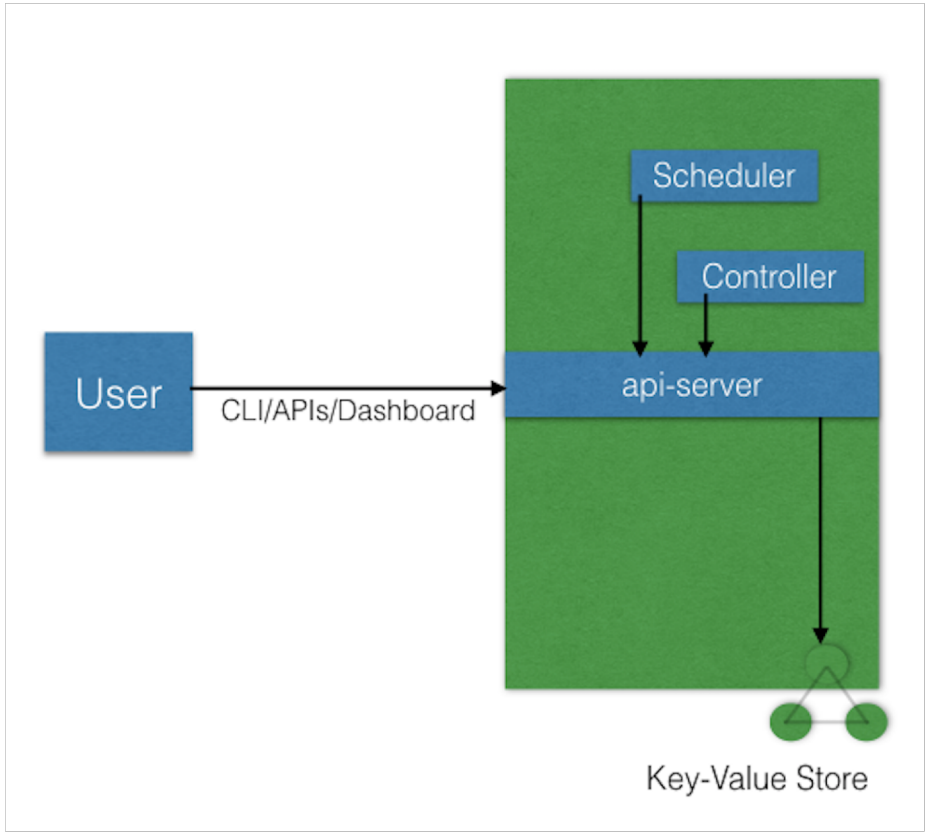
For fault tolerance purposes, there can be more than one master node in the cluster. If we have more than one master node, they would be in a HA (High Availability) mode, and only one of them will be the leader, performing all the operations. The rest of the master nodes would be followers.

To manage the cluster state, Kubernetes uses etcd, and all master nodes connect to it. **etcd** is a distributed key-value store, which we will discuss in a little bit. The key-value store can be part of the master node. It can also be configured externally, in which case, the master nodes would connect to it.

Master Node Components

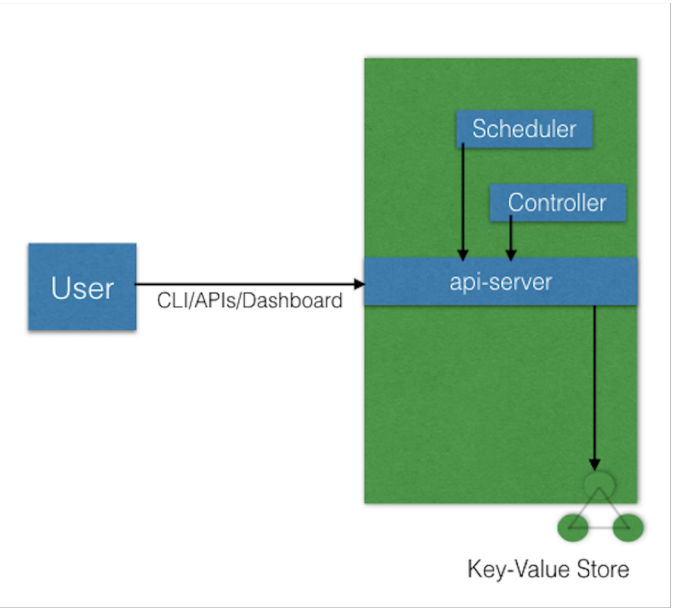
A master node has the following components:

- API server #เด็กรับออเดอร์
- Scheduler #คนจ่ายง่าย
- Controller manager #คนคุมงาน
- etcd. #กระดาษจดบันทึก



Master Node Components: API Server

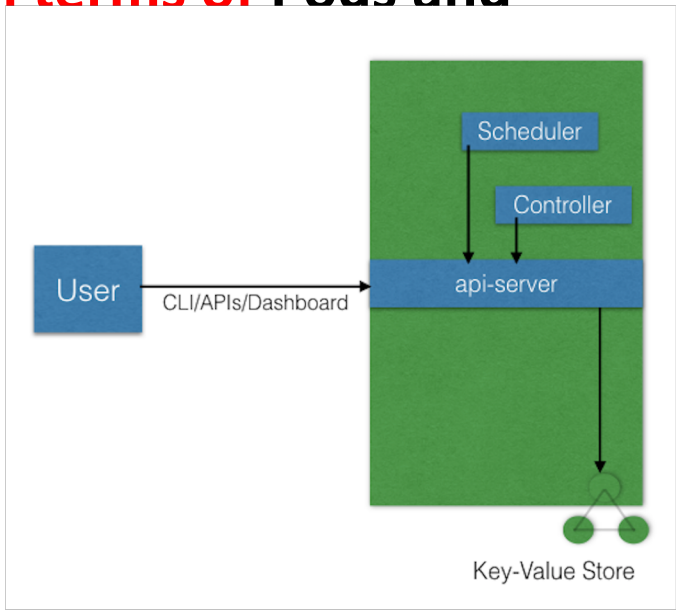
All the administrative tasks are performed via the **API server** within the master node. A user/operator sends **REST commands to the API server**, which then validates and processes the requests. After executing the requests, the resulting state of the cluster is stored in the distributed key-value store.



Master Node Components: Scheduler

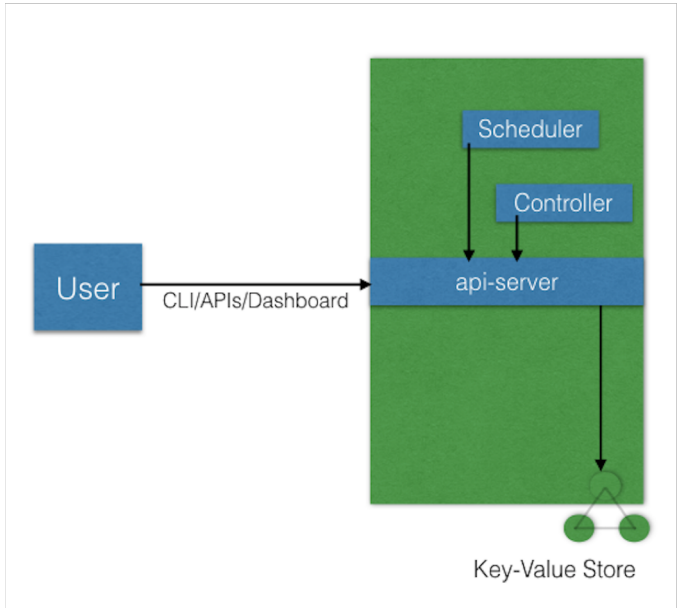
As the name suggests, the **scheduler** **schedules the work to different worker nodes.**

The scheduler **has the resource usage information for each worker node**. It also knows about the **constraints that users/operators may have set**, such as scheduling work on a node that has the label **`disk==ssd`** set. Before scheduling the work, the scheduler also takes into account the quality of the service requirements, data locality, affinity, anti-affinity, etc. The **scheduler schedules the work in terms of Pods and Services.**



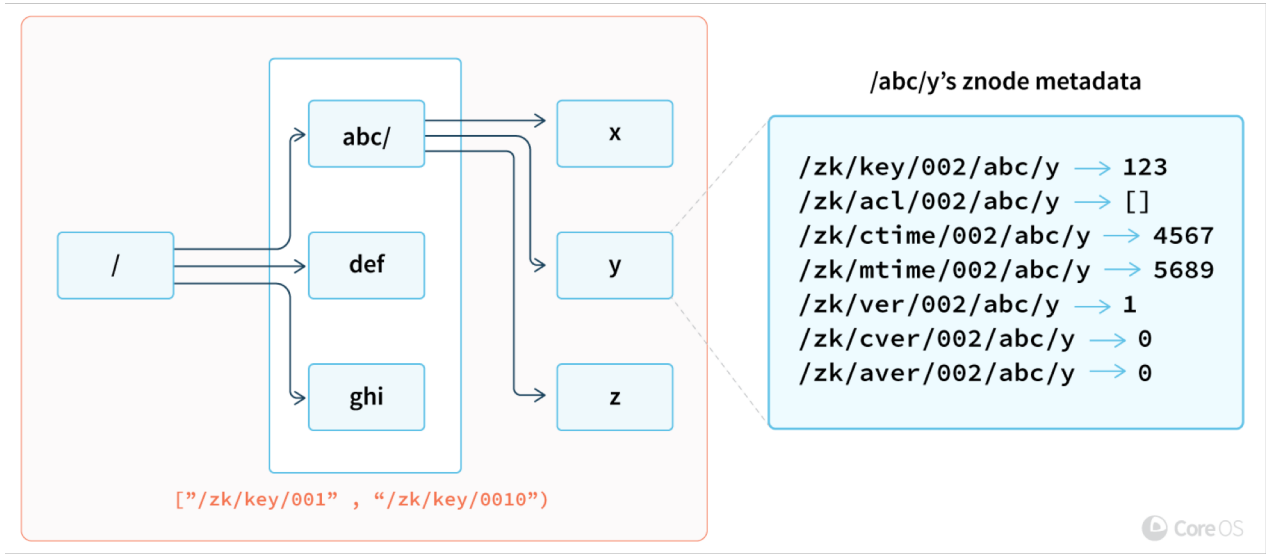
Master Node Components: Controller Manager

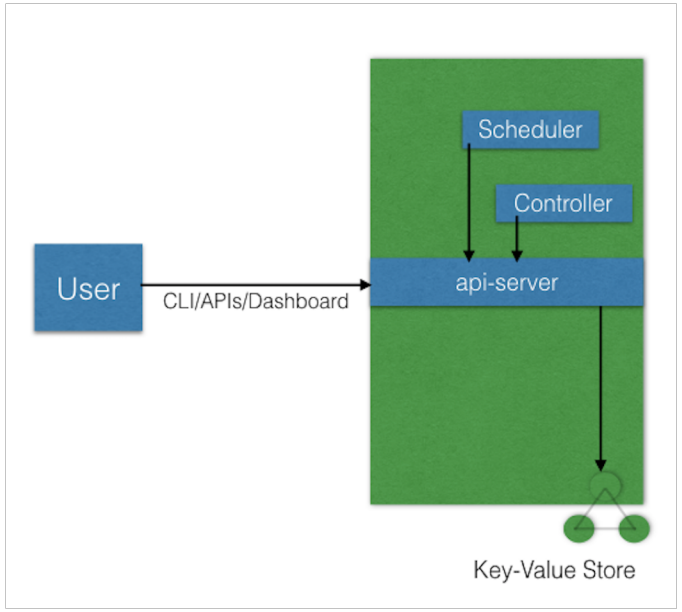
The controller manager manages different **non-terminating control loops**, which **regulate the state of the Kubernetes cluster**. Each one of these control loops knows about the **desired state** of the objects it manages, and watches their **current state** through the API server. In a control loop, if the current state of the objects it manages does not meet the desired state, then the control loop takes corrective steps to make sure that the **current state** is the same as the **desired state**.



Master Node Components: etcd

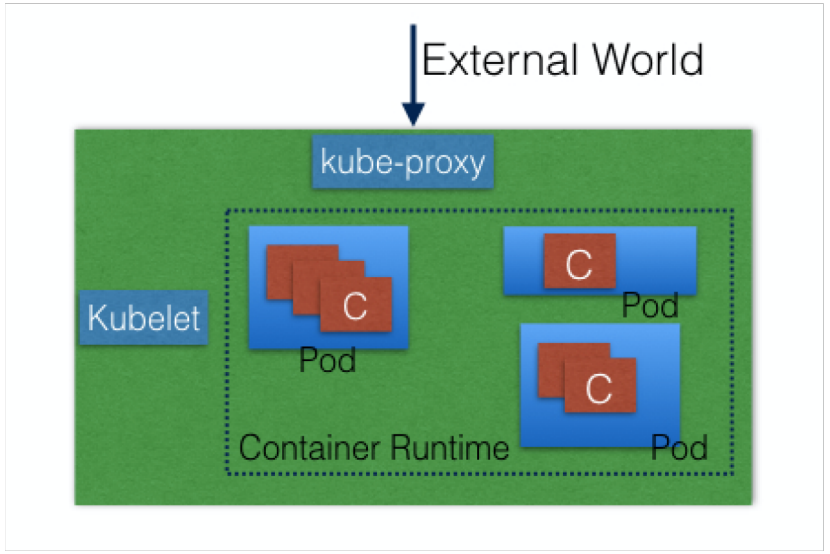
As discussed earlier, **etcd** is a **distributed key-value store** which is used to **store the cluster state**. It can be part of the Kubernetes Master, or, it can be configured externally, in which case, master nodes would connect to it.





Worker Node

A **worker node** is a machine (VM, physical server, etc.) which **runs the applications using Pods and is controlled by the master node**. **Pods are scheduled on the worker nodes**, which have the necessary tools to run and connect them. **A Pod is the scheduling unit** in Kubernetes. It is a logical collection of one or more containers which are always scheduled together.

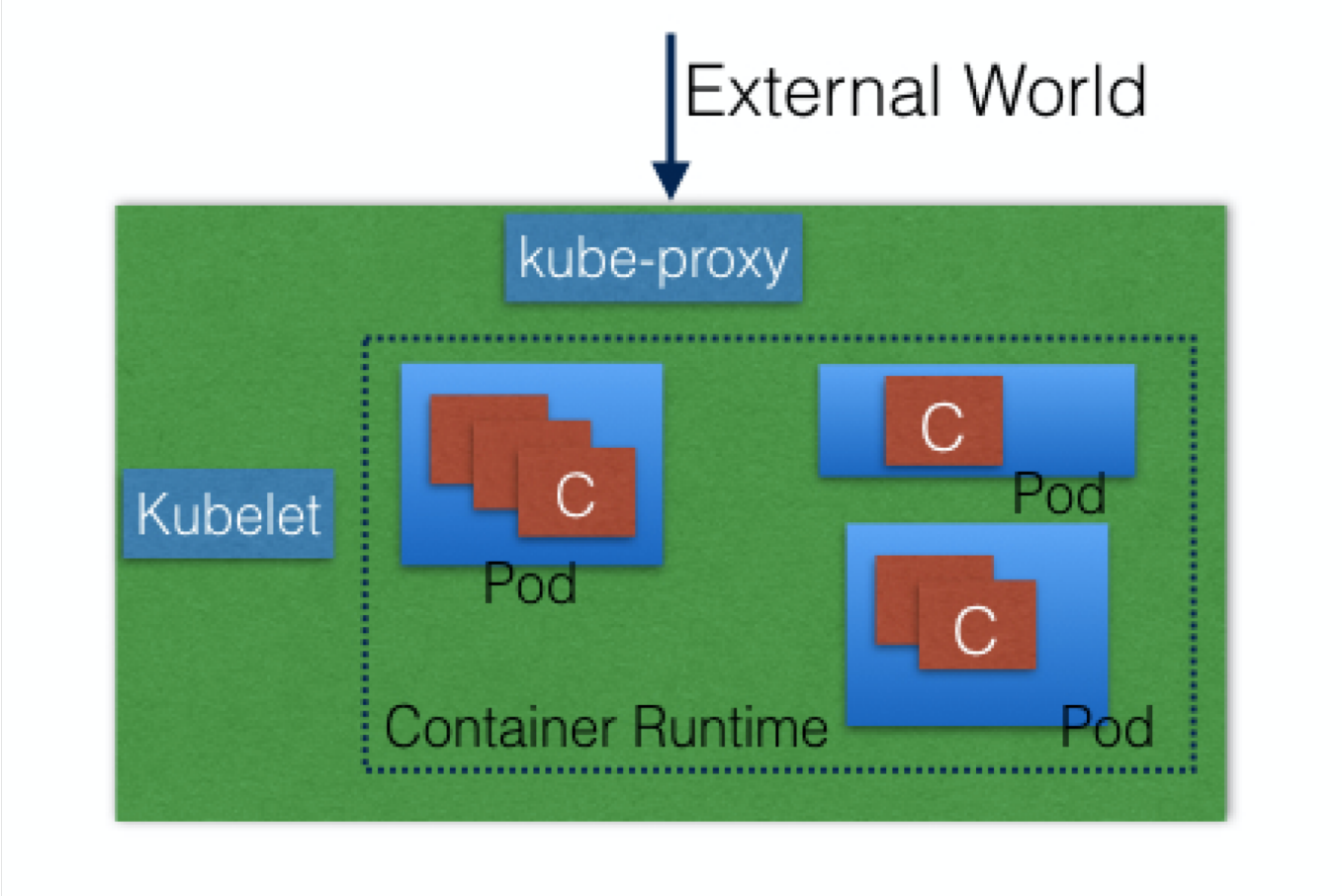


Also**, to access the applications from the external world, we connect to worker nodes and not to the master node/s**.

Worker Node Components

A worker node has the following components:

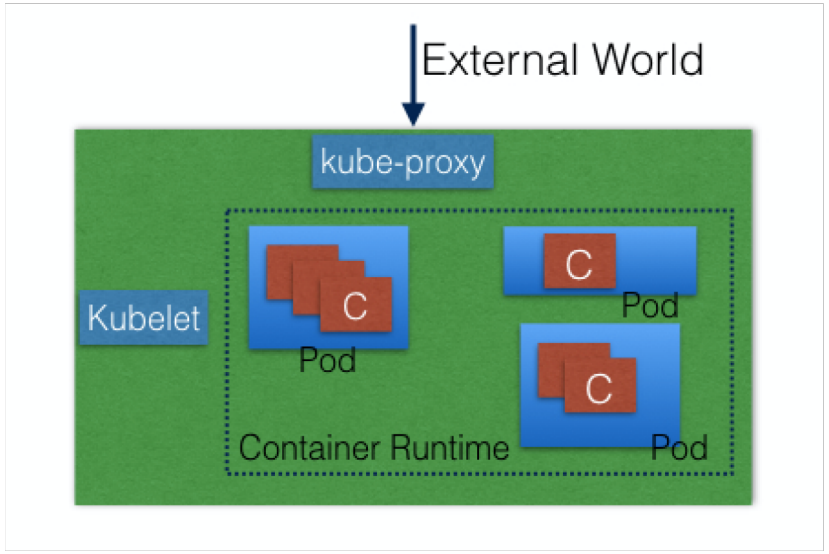
- Container runtime
- kubelet
- kube-proxy.



Worker Node Components: Container Runtime

To run and manage a container's lifecycle, we need a **container runtime** on the worker node. Some examples of container runtimes are:

- containerd
- rkt
- lxd.

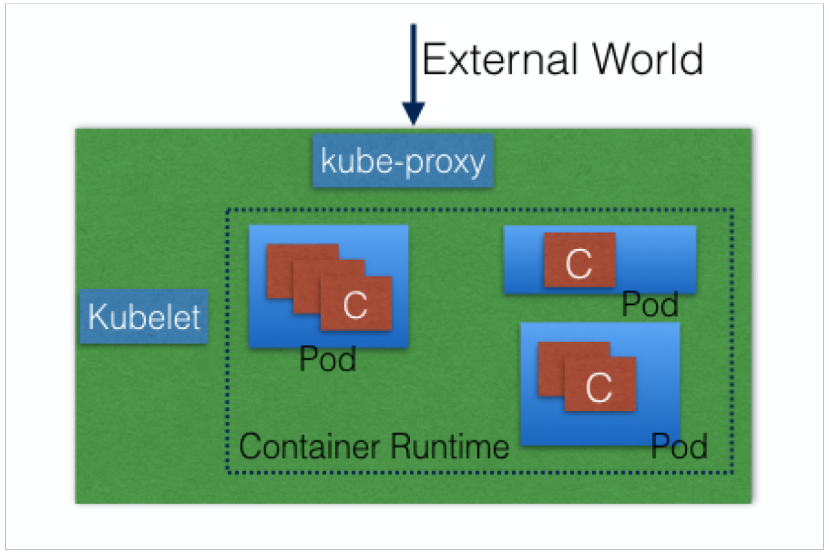


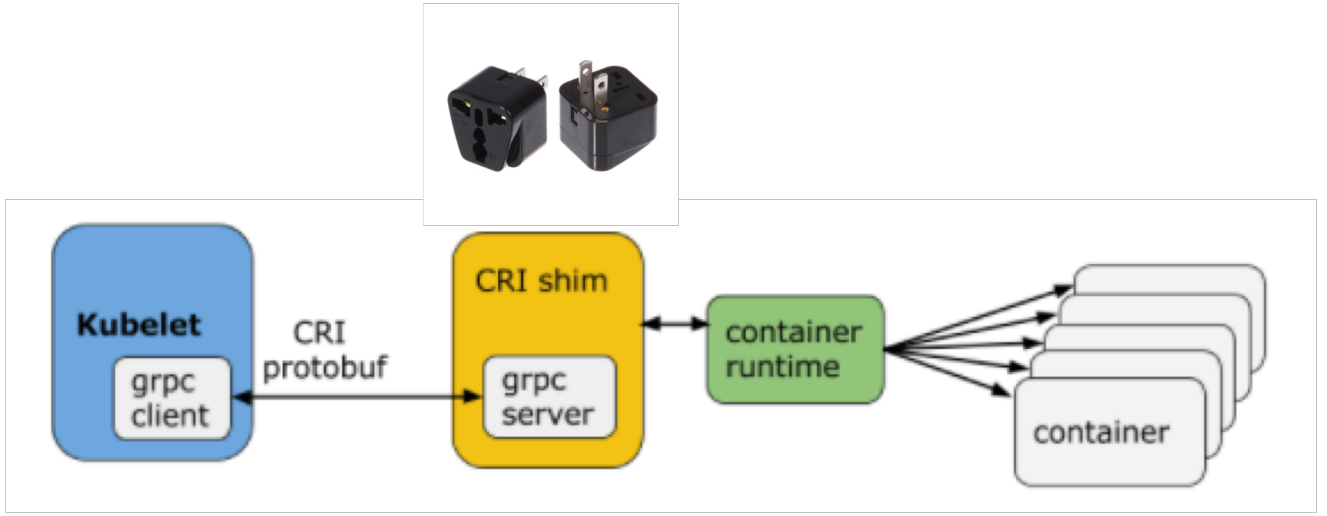
Sometimes, Docker is also referred to as a container runtime, but to be precise, Docker is a platform which uses **containerd** as a container runtime.

Worker Node Components: kubelet

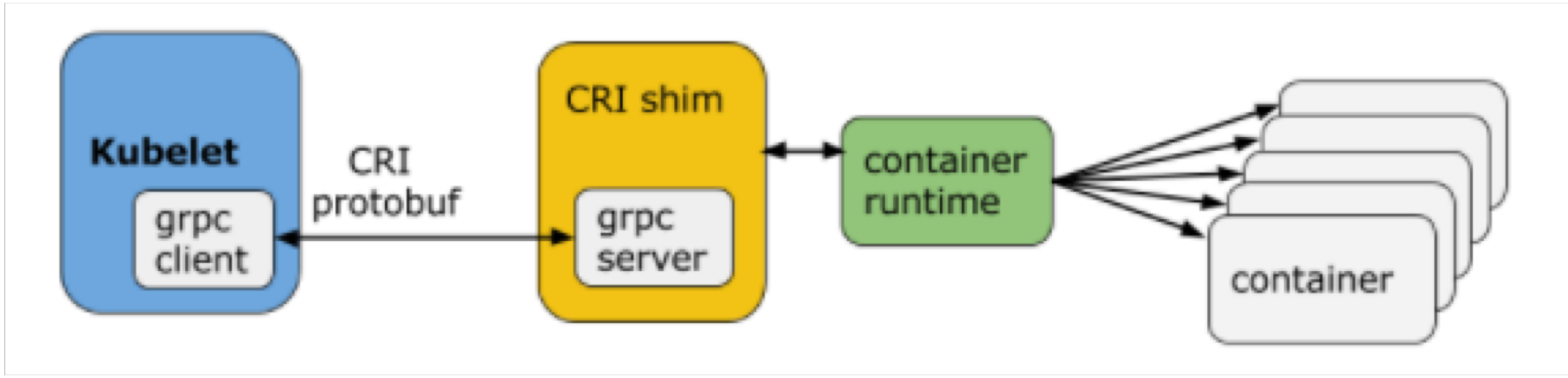
The **kubelet** is an **agent** which **runs on each worker node and communicates with the master node**. It receives the Pod definition via various means (primarily, through the API server), and runs the containers associated with the Pod. It also makes sure that the containers which are part of the Pods are healthy at all times.

The kubelet connects to the container runtime using **Container Runtime Interface (CRI).** The Container Runtime Interface consists of protocol buffers, gRPC API, and libraries.





Worker Node Components: kubelet



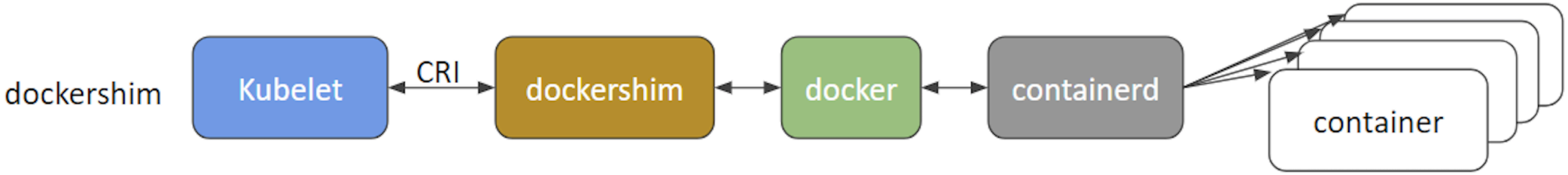
As shown above, the kubelet (grpc client) connects to the CRI shim (grpc server) to perform container and image operations. CRI implements two services: **`ImageService`** and **`RuntimeService`**. The **`ImageService`** is responsible for all the **image-related operations**, while the **`RuntimeService`** is responsible for all the **Pod and container-related** operations.

Container runtimes used to be hard-coded in Kubernetes, but with the development of CRI, Kubernetes can now use different container runtimes without the need to recompile. Any container runtime that implements CRI can be used by Kubernetes to manage Pods, containers, and container images.

Worker Node Components: kubelet: CRI shims

Dockershim

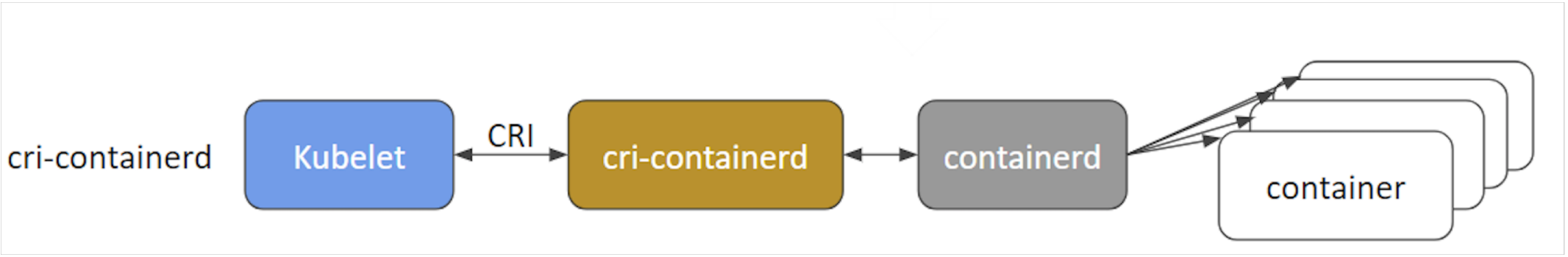
With dockershim, containers are created using Docker installed on the worker nodes. Internally, Docker uses containerd to create and manage containers.



Worker Node Components: kubelet: CRI shims

Below you will find some examples of CRI shims:

cri-containerd
With cri-containerd, we can directly use Docker's smaller offspring containerd to create and manage containers.



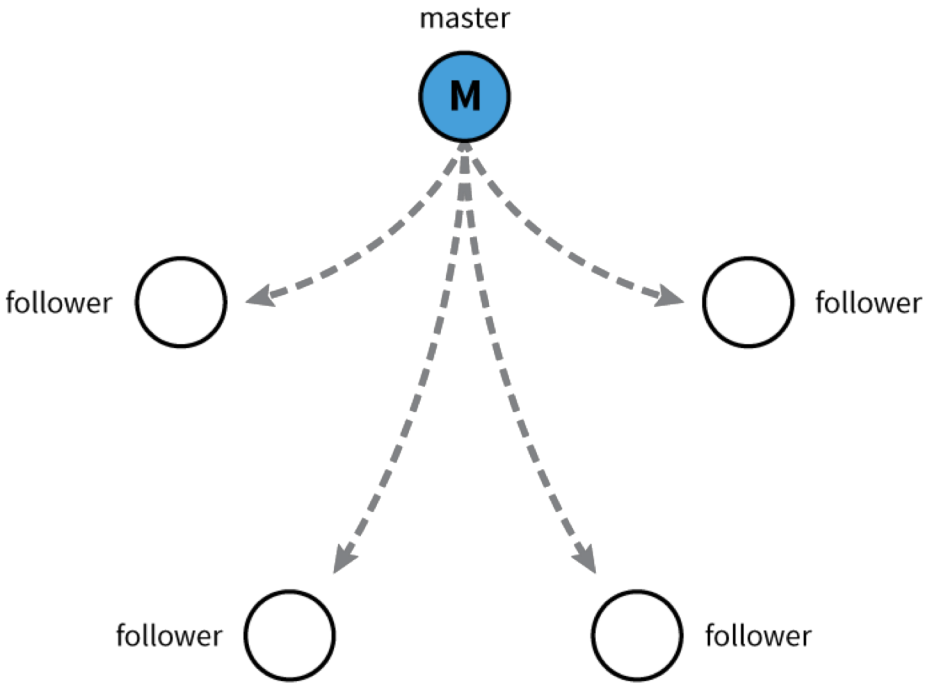
Worker Node Components: kube-proxy

Instead of connecting directly to **Pods** to access the applications, we use a logical construct called a **Service** as a connection endpoint. **A Service groups related Pods and, when accessed, load balances to them**. We will talk more about Services in later chapters.

kube-proxy is the network proxy which runs on each worker node and **listens to the API server for each Service endpoint creation/deletion**. For each Service endpoint, kube-proxy sets up the routes so that it can reach to it. We will also explore this in more detail in later chapters.

State Management with etcd

As we mentioned earlier, Kubernetes uses **etcd** to store the cluster state. etcd is a distributed key-value store based on the Raft Consensus Algorithm. Raft allows a collection of machines to work as a coherent group that can survive the failures of some of its members. At any given time, **one of the nodes in the group will be the master, and the rest of them will be the followers**. Any node can be treated as a master.



etcd is written in the Go programming language. In Kubernetes, besides storing the cluster state, etcd is also used to store configuration details such as **subnets, ConfigMaps, Secrets, etc.**

Network Setup Challenges

To have a fully functional Kubernetes cluster, we need to make sure of the following:

- A unique IP is assigned to each Pod
- Containers in a Pod can communicate to each other
- The Pod is able to communicate with other Pods in the cluster
- If configured, the application deployed inside a Pod is accessible from the external world.

All of the above are networking challenges which must be addressed before deploying the Kubernetes cluster.

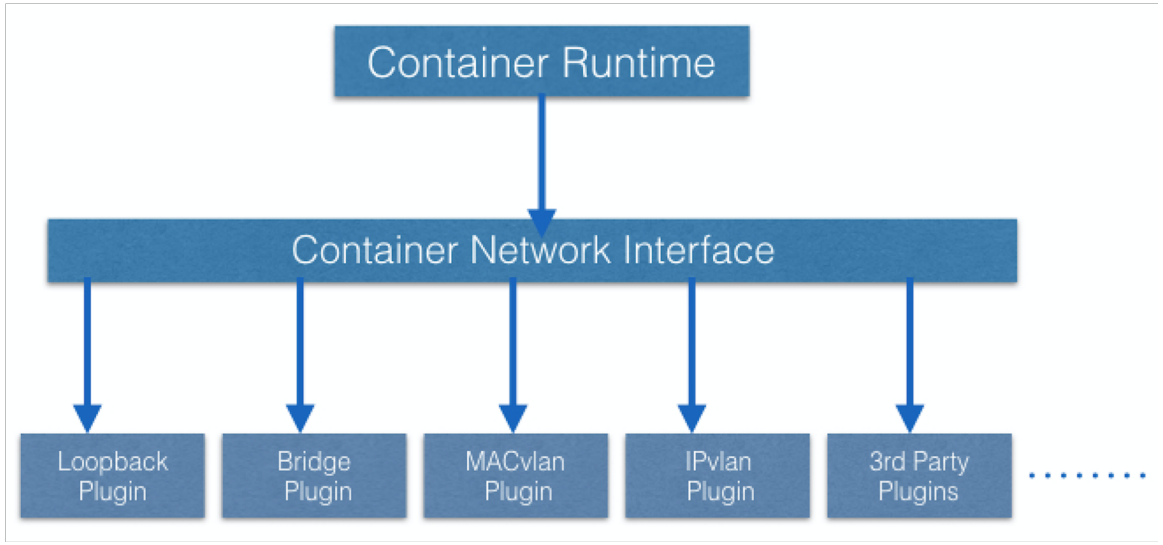
Assigning a Unique IP Address to Each Pod

In Kubernetes, each Pod gets a unique IP address. For container networking, there are two primary specifications:

- **Container Network Model (CNM)**, proposed by Docker
- **Container Network Interface (CNI)**, proposed by CoreOS.

Kubernetes uses CNI to assign the IP address to each Pod.

The container runtime **offloads the IP assignment to CNI,** which connects to the underlying configured plugin, like Bridge or MACvlan, to get the IP address. Once the IP address is given by the respective plugin, CNI forwards it back to the requested container runtime.



Container Network Interface (CNI)

Container-to-Container Communication Inside a Pod

- With the help of the underlying host operating system, all of the container runtimes generally create an isolated network entity for each container that it starts. On Linux, that entity is referred to as a **network namespace**. These network namespaces can be shared across containers, or with the host operating system.

- Inside a Pod, containers share the network namespaces, so that they can reach to each other via localhost.

Pod-to-Pod Communication Across Nodes

In a clustered environment, the Pods can be scheduled on any node. We need to make sure that the Pods can communicate across the nodes, and all the nodes should be able to reach any Pod. Kubernetes also puts a condition that there shouldn't be any Network Address Translation (NAT) while doing the Pod-to-Pod communication across hosts. We can achieve this via:

- Routable Pods and nodes, using the underlying physical infrastructure, like Google Kubernetes Engine
- Using Software Defined Networking, like Flannel, Weave, Calico, etc.

Communication Between the External World and Pods

By exposing our services to the external world with **kube-proxy**, we can access our applications from outside the cluster.

Learning Objectives (Review)

You should now be able to:

- Discuss the Kubernetes architecture.
- Explain the different components for master and worker nodes.
- Discuss about cluster state management with etcd.
- Review the Kubernetes network setup requirements.