2.22.a. CNI Network Configuration File

To provide container networking, **Kubernetes** is standardizing on the Container Network Interface (CNI) specification. As of v1.6.0, **kubeadm** (the **Kubernetes** cluster bootstrapping tool) uses CNI as the default network interface mechanism.

CNI is an emerging specification with associated libraries to write plugins that configure container networking and remove allocated resources when the container is deleted. Its aim is to provide a common interface between the various networking solutions and container runtimes. As the CNI specification is language-agnostic, there are many plugins from Amazon ECS, to SR-IOV, to Cloud Foundry, and more.

2.22.b. CNI Network Configuration File (Cont.)

With CNI, you can write a network configuration file:

```
{
    "cniVersion": "0.2.0",
    "name": "mynet",
    "type": "bridge",
    "bridge": "cni0",
    "isGateway": true,
    "ipMasq": true,
    "ipam": {
        "type": "host-local",
        "subnet": "10.22.0.0/16",
        "routes": [
            { "dst": "0.0.0.0/0" }
         ]
    }
}
```

This configuration defines a standard **Linux** bridge named `cni0`, which will give out IP addresses in the subnet `10.22.0.0./16`. The `bridge` plugin will configure the network interfaces in the correct namespaces to define the container network properly.

The main `README` of the CNI GitHub repository has more information.

2.23. Pod-to-Pod Communication

While a CNI plugin can be used to configure the network of a pod and provide a single IP per pod, CNI does not help you with pod-to-pod communication across nodes.

The requirement from **Kubernetes** is the following:

- All pods can communicate with each other across nodes.
- All nodes can communicate with all pods.
- No Network Address Translation (NAT).

Basically, all IPs involved (nodes and pods) are routable without NAT. This can be achieved at the physical network infrastructure if you have access to it (e.g. GKE). Or, this can be achieved with a software defined overlay with solutions like:

- Weave
- Flannel
- Calico
- Romana

See the cluster networking documentation page or the list of networking add-ons for a more complete list.

2.24. The Cloud Native Computing Foundation

Kubernetes is an open source software with an Apache license. Google donated **Kubernetes** to a newly formed collaborative project within **The Linux Foundation** in July 2015, when **Kubernetes** reached the v1.0 release. This project is known as the **Cloud Native Computing Foundation** (**CNCF**).

CNCF is not just about **Kubernetes**, it serves as the governing body for open source software that solves specific issues faced by cloud native applications (i.e. applications that are written specifically for a cloud environment).

CNCF has many corporate members that collaborate, such as Cisco, the Cloud Foundry Foundation, AT&T, Box, Goldman Sachs, and many others.

Note: Since CNCF now owns the **Kubernetes** copyright, contributors to the source need to sign a contributor license agreement (CLA) with CNCF, just like any contributor to an Apache-licensed project signs a CLA with the **Apache Software Foundation**.

CLOUD NATIVE
COMPUTING
FOUNDATION

2.25. Resource Recommendations

If you want to go beyond this general introduction to **Kubernetes**, here are a few things we recommend:

- Read the Borg paper.
- Listen to John Wilkes talking about Borg and Kubernetes.
- Add the **Kubernetes** community hangout to your calendar, and attend at least once.
- Join the community on Slack and go in the **#kubernetes-users** channel.
- Check out the very active Stack Overflow community.

2.1. Kubernetes Architecture

CHAPTER 2 KUBERNETES ARCHITECTURE

2.1. Kubernetes Architecture

CHAPTER 2
KUBERNETES ARCHITECTURE

2.3. Learning Objectives

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

- Discuss the history of Kubernetes.
- Learn about master node components.
- Learn about minion (worker) node components.
- Understand the Container Network Interface (CNI) configuration and Network Plugins.

2.4. What Is Kubernetes?

Running a container on a laptop is relatively simple. But, connecting containers across multiple hosts, scaling them, deploying applications without downtime, and service discovery among several aspects, can be difficult.

Kubernetes addresses those challenges from the start with a set of primitives and a powerful open and extensible API. The ability to add new objects and controllers allows easy customization for various production needs.

According to the kubernetes.io website, **Kubernetes** is:

"an open-source system for automating deployment, scaling, and management of containerized applications".

A key aspect of Kubernetes is that it builds on 15 years of experience at Google in a project called **borg**.

Google's infrastructure started reaching high scale before virtual machines became pervasive in the datacenter, and containers provided a fine-grained solution for packing clusters efficiently. Efficiency in using clusters and managing distributed applications has been at the core of Google challenges.

In Greek, **κνβερνητης** means *the Helmsman*, or pilot of the ship. Keeping with the maritime theme of Docker containers, Kubernetes is the pilot of a ship of containers.

Due to the difficulty in pronouncing the name, many will use a nickname, **K8s**, as Kubernetes has eight letters between K and S. The nickname is said like **Kate's**.

2.5. Components of Kubernetes

Deploying containers and using Kubernetes may require a change in the development and the system administration approach to deploying applications. In a traditional environment, an application (such as a web server) would be a monolithic application placed on a dedicated server. As the web traffic increases, the application would be tuned, and perhaps moved to bigger and bigger hardware. After a couple of years, a lot of customization may have been done in order to meet the current web traffic needs.

Instead of using a large server, Kubernetes approaches the same issue by deploying a large number of small web servers, or **microservices**. The server and client sides of the application expect that there are many possible agents available to respond to a request. It is also important that clients expect the server processes to die and be replaced, leading to a transient server deployment. Instead of a large **Apache** web server with many **httpd** daemons responding to page requests, there would be many **nginx** servers, each responding.

The transient nature of smaller services also allows for decoupling. Each aspect of the traditional application is replaced with a dedicated, but transient, microservice or agent. To join these agents, or their replacements together, we use **services** and API calls. A service ties traffic from one agent to another (for example, a frontend web server to a backend database) and handles new IP or other information, should either one die and be replaced.

Communication to, as well as internally, between components is API call-driven, which allows for flexibility. Configuration information is stored in a **JSON** format, but is most often written in **YAML**. Kubernetes agents convert the YAML to JSON prior to persistence to the database.

Kubernetes is written in **Go Language**, a portable language which is like a hybridization between C++, Python, and Java. Some claim it incorporates the best (while some claim the worst) parts of each.

2.6. Challenges

Containers have seen a huge rejuvenation in the past three years. They provide a great way to package, ship, and run applications - that is the **Docker** motto.

The developer experience has been boosted tremendously thanks to containers. Containers, and **Docker** specifically, have empowered developers with ease of building container images, simplicity of sharing images via **Docker** registries, and providing a powerful user experience to manage containers.

However, managing containers at scale and architecting a distributed application based on microservices' principles is still challenging.

You first need a continuous integration pipeline to build your container images, test them, and verify them. Then, you need a cluster of machines acting as your base infrastructure on which to run your containers. You also need a system to launch your containers, and watch over them when things fail and self-heal. You must be able to perform rolling updates and rollbacks, and eventually tear down the resource when no longer needed.

All of these actions require flexible, scalable, and easy-to-use network and storage. As containers are launched on any worker node, the network must join the resource to other containers, while still keeping the traffic secure from others. We also need a storage structure which provides and keeps or recycles storage in a seamless manner.

2.7.a. The Borg Heritage

What primarily distinguishes **Kubernetes** from other systems is its heritage. **Kubernetes** is inspired by **Borg** - the internal system used by Google to manage its applications (e.g. **Gmail**, **Apps**, **GCE**).

With Google pouring the valuable lessons they learned from writing and operating **Borg** for over 15 years into **Kubernetes**, this makes **Kubernetes** a safe choice when having to decide on what system to use to manage containers. While a powerful tool, part of the current growth in Kubernetes is making it easier to work with and handle workloads not found in a Google data center.

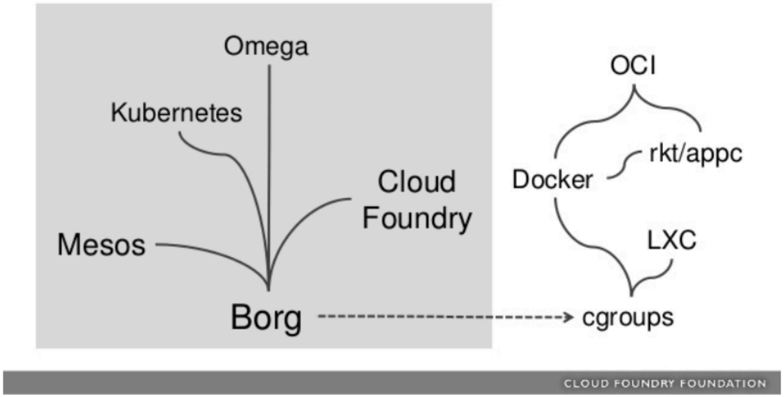
To learn more about the ideas behind Kubernetes, you can read the *Large-scale cluster management at Google with Borg paper*.

Borg has inspired current data center systems, as well as the underlying technologies used in container runtime today. **Google** contributed **cgroups** to the Linux kernel in 2007; it limits the resources used by collection of processes. Both **cgroups** and **Linux namespaces** are at the heart of containers today, including **Docker**.

Mesos was inspired by discussions with **Google** when **Borg** was still a secret. Indeed, **Mesos** builds a multi-level scheduler, which aims to better use a data center cluster.

The Cloud Foundry Foundation embraces the 12 factor application principles. These principles provide great guidance to build web applications that can scale easily, can be deployed in the cloud, and whose build is automated. **Borg** and **Kubernetes** address these principles as well.

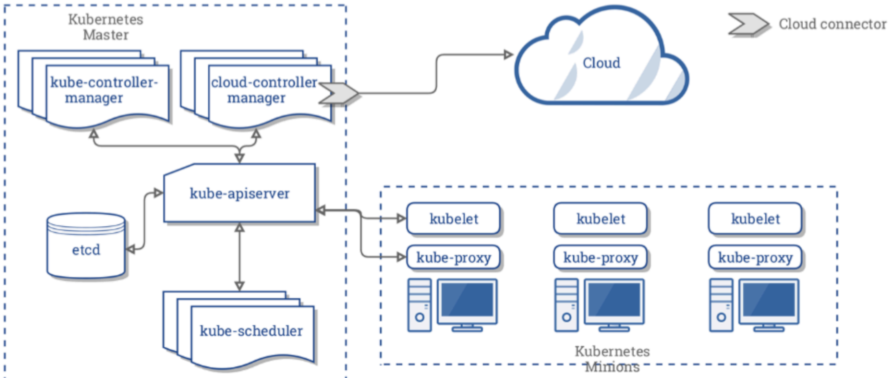
2.7.b. The Borg Heritage (Cont.)



The Kubernetes Lineage (by Chip Childers, Cloud Foundry Foundation, retrieved from LinkedIn Slideshare)

2.8.a. Kubernetes Architecture

To quickly demistify **Kubernetes**, let's have a look at the *Kubernetes Architecture* graphic, which shows a high-level architecture diagram of the system components.



Kubernetes Architecture

2.8.b. Kubernetes Architecture (Cont.)

In its simplest form, **Kubernetes** is made of a central manager (aka master) and some worker nodes (we will see in a follow-on chapter how you can actually run everything on a single node for testing purposes). The manager runs an API server, a scheduler, various controllers and a storage system to keep the state of the cluster, container settings, and the networking configuration.

Kubernetes exposes an API (via the API server): you can communicate with the API using a local client called **kubectl** or you can write your own client. The **kube-scheduler** sees the requests for running containers coming to the API and finds a suitable node to run that container in. Each node in the cluster runs two processes: a **kubelet** and a **kube-proxy**. The **kubelet** receives requests to run the containers, manages any necessary resources and watches over them on the local node. The **kube-proxy** creates and manages networking rules to expose the container on the network.

2.9. Terminology

We have learned that Kubernetes is an orchestration system to deploy and manage containers. Containers are not managed individually; instead, they are part of a larger object called a **Pod**. A **Pod** consists of one or more containers which share an IP address, access to storage and namespace. Typically, one container in a Pod runs an application, while other containers support the primary application.

Orchestration is managed through a series of watch-loops, or **controllers**. Each controller interrogates the **kube-apiserver** for a particular object state, modifying the object until the declared state matches the current state. The default, newest, and feature-filled controller for containers is a **Deployment**. A **Deployment** ensures that resources are available, such as IP address and storage, and then deploys a **ReplicaSet**. The **ReplicaSet** is a controller which deploys and restarts containers, Docker by default, until the requested number of containers is running. Previously, the function was handled by the **ReplicationController**, but has been obviated by **Deployments**. There are also **Jobs** and **CronJobs** to handle single or recurring tasks.

To easily manage thousands of Pods across hundreds of nodes can be a difficult task to manage. To make management easier, we can use **labels**, arbitrary strings which become part of the object metadata. These can then be used when checking or changing the state of objects without having to know individual names or UIDs. Nodes can have **taints** to discourage Pod assignments, unless the Pod has a **toleration** in its metadata.

There is also space in metadata for **annotations** which remain with the object but cannot be used by Kubernetes commands. This information could be used by third-party agents or other tools.

2.10. Master Node

The Kubernetes master runs various server and manager processes for the cluster. Among the components of the master node are the **kube-apiserver**, the **kube-scheduler**, and the **etcd** database. As the software has matured, new components have been created to handle dedicated needs, such as the **cloud-controller-manager**; it handles tasks once handled by the **kube-controller-manager** to interact with other tools, such as **Rancher** or **DigitalOcean** for third-party cluster management and reporting.

There are several add-ons which have become essential to a typical production cluster, such as DNS services. Others are third-party solutions where Kubernetes has not yet developed a local component, such as cluster-level logging and resource monitoring.

2.11. kube-apiserver

The **kube-apiserver** is central to the operation of the Kubernetes cluster.

All calls, both internal and external traffic, are handled via this agent. All actions are accepted and validated by this agent, and it is the only connection to the **etcd** database. As a result, it acts as a master process for the entire cluster, and acts as a frontend of the cluster's shared state.

2.12. kube-scheduler

The **kube-scheduler** uses an algorithm to determine which node will host a Pod of containers. The scheduler will try to view available resources (such as volumes) to bind, and then try and retry to deploy the Pod based on availability and success.

There are several ways you can affect the algorithm, or a custom scheduler could be used instead. You can also bind a Pod to a particular node, though the Pod may remain in a pending state due to other settings.

One of the first settings referenced is if the Pod can be deployed within the current quota restrictions. If so, then the taints and tolerations, and labels of the Pods are used along with those of the nodes to determine the proper placement.

You can find more details about the scheduler can be found on GitHub.

2.13. etcd Database

The state of the cluster, networking, and other persistent information is kept in an **etcd** database, or, more accurately, a *b+tree* key-value store. Rather than finding and changing an entry, values are always appended to the end. Previous copies of the data are then marked for future removal by a compaction process. It works with **curl** and other HTTP libraries, and provides reliable watch queries.

Simultaneous requests to update a value all travel via the **kube-apiserver**, which then passes along the request to **etcd** in a series. The first request would update the database. The second request would no longer have the same version number, in which case the **kube-apiserver** would reply with an error 409 to the requester. There is no logic past that response on the server side, meaning the client needs to expect this and act upon the denial to update.

There is a **`master`** database along with possible **`followers`**. They communicate with each other on an ongoing basis to determine which will be **master**, and determine another in the event of failure. While very fast and potentially durable, there have been some hiccups with new tools, such as **kubeadm**, and features like whole cluster upgrades.

2.14. Other Agents

The **kube-controller-manager** is a core control loop daemon which interacts with the **kube-apiserver** to determine the state of the cluster. If the state does not match, the manager will contact the necessary controller to match the desired state. There are several controllers in use, such as *endpoints*, *namespace*, and *replication*. The full list has expanded as Kubernetes has matured.

In **alpha** since v1.8, the **cloud-controller-manager** interacts with agents outside of the cloud. It handles tasks once handled by **kube-controller-manager**. This allows faster changes without altering the core Kubernetes control process. Each **kubelet** must use the `--cloud-provider-external` settings passed to the binary.

2.15. Minion (Worker) Nodes

All worker nodes run the **kubelet** and **kube-proxy**, as well as the container engine, such as **Docker** or **rkt**. Other management daemons are deployed to watch these agents or provide services not yet included with Kubernetes.

The **kubelet** interacts with the underlying Docker Engine also installed on all the nodes, and makes sure that the containers that need to run are actually running. The **kube-proxy** is in charge of managing the network connectivity to the containers. It does so through the use of **iptables** entries. It also has the `userspace` mode, in which it monitors *Services* and *Endpoints* using a random port to proxy traffic and an alpha feature of `ipvs`.

You can also run an alternative to the **Docker** engine: **rkt** by **CoreOS**. To learn how you can do that, you should check the documentation. In future releases, it is highly likely that Kubernetes will support additional container runtime engines.

Supervisord is a lightweight process monitor used in traditional Linux environments to monitor and notify about other processes. In the cluster, this daemon monitors both the **kubelet** and **docker** processes. It will try to restart them if they fail, and log events.

Kubernetes does not have cluster-wide logging yet. Instead, another **CNCF** project is used, called Fluentd. When implemented, it provides a unified logging layer for the cluster, which filters, buffers, and routes messages.

2.15. Minion (Worker) Nodes

All worker nodes run the **kubelet** and **kube-proxy**, as well as the container engine, such as **Docker** or **rkt**. Other management daemons are deployed to watch these agents or provide services not yet included with Kubernetes.

The **kubelet** interacts with the underlying Docker Engine also installed on all the nodes, and makes sure that the containers that need to run are actually running. The **kube-proxy** is in charge of managing the network connectivity to the containers. It does so through the use of **iptables** entries. It also has the `userspace` mode, in which it monitors *Services* and *Endpoints* using a random port to proxy traffic and an alpha feature of `ipvs`.

You can also run an alternative to the **Docker** engine: **rkt** by **CoreOS**. To learn how you can do that, you should check the documentation. In future releases, it is highly likely that Kubernetes will support additional container runtime engines.

Supervisord is a lightweight process monitor used in traditional Linux environments to monitor and notify about other processes. In the cluster, this daemon monitors both the **kubelet** and **docker** processes. It will try to restart them if they fail, and log events.

Kubernetes does not have cluster-wide logging yet. Instead, another **CNCF** project is used, called Fluentd. When implemented, it provides a unified logging layer for the cluster, which filters, buffers, and routes messages.

2.16. kubelet

The **kubelet** agent is the heavy lifter for changes and configuration on worker nodes. It accepts the API calls for Pod specifications (a `PodSpec` is a JSON or YAML file that describes a pod). It will work to configure the local node until the specification has been met.

Should a Pod require access to storage, *Secrets* or *ConfigMaps*, the **kubelet** will ensure access or creation. It also sends back status to the **kube-apiserver** for eventual persistence.

2.17. Pods

The whole point of Kubernetes is to orchestrate the lifecycle of a container. We do not interact with particular containers. Instead, the smallest unit we can work with is a *Pod*. Some would say a pod of whales or peas-in-a-pod. Due to shared resources, the design of a *Pod* typically follows a one-process-per-container architecture.

Containers in a *Pod* are started in parallel. As a result, there is no way to determine which container becomes available first inside a pod. To support a single process running in a container, you may need logging, a proxy, or special adapter. These tasks are often handled by other containers in the same pod.

There is only one IP address per Pod. If there is more than one container, they must share the IP. To communicate with each other, they can either use IPC, or a shared filesystem.

While Pods are often deployed with one application container in each, a common reason to have multiple containers in a Pod is for logging. You may find the term **`sidecar`** for a container dedicated to performing a helper task, like handling logs and responding to requests, as the primary application container may have this ability.

2.18. Services

With every object and agent decoupled we need a flexible and scalable agent which connects resources together and will reconnect, should something die and a replacement is spawned. Each *Service* is a microservice handling a particular bit of traffic, such as a single **`NodePort`** or a **`LoadBalancer`** to distribute inbound requests among many Pods.

A *Service* also handles access policies for inbound requests, useful for resource control, as well as for security.

2.19. Controllers

An important concept for orchestration is the use of controllers. Various controllers ship with Kubernetes, and you can create your own, as well. A simplified view of a controller is an agent, or **`Informer`**, and a downstream store. Using a **`DeltaFIFO`** queue, the source and downstream are compared. A loop process receives an **`obj`** or object, which is an array of deltas from the FIFO queue. As long as the delta is not of the type **`Deleted`**, the logic of the controller is used to create or modify some object until it matches the specification.

The **`Informer`** which uses the API server as a source requests the state of an object via an API call. The data is cached to minimize API server transactions. A similar agent is the **`SharedInformer`**; objects are often used by multiple other objects. It creates a shared cache of the state for multiple requests.

A **`Workqueue`** uses a key to hand out tasks to various workers. The standard **Go** work queues of rate limiting, delayed, and time queue are typically used.

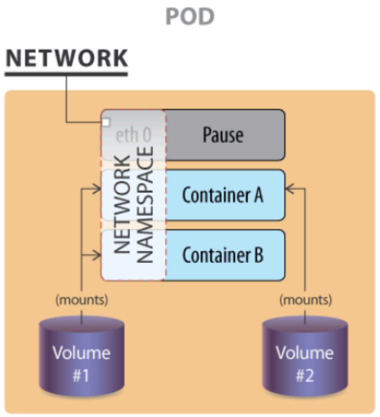
The **`endpoints`**, **`namespace`**, and **`serviceaccounts`** controllers each manage the eponymous resources for Pods.

2.20. Single IP per Pod

A pod represents a group of co-located containers with some associated data volumes. All containers in a pod share the same network **namespace**.

The graphic shows a pod with two containers, A and B, and two data volumes, 1 and 2. Containers A and B share the network namespace of a third container, known as the ***pause* container**. The pause container is used to get an IP address, then all the containers in the pod will use its network namespace. Volumes 1 and 2 are shown for completeness.

To communicate with each other, Pods can use the loopback interface, write to files on a common filesystem, or via inter-process communication (IPC).



2.21. Networking Setup

Getting all the previous components running is a common task for system administrators who are accustomed to configuration management. But, to get a fully functional **Kubernetes** cluster, the network will need to be set up properly, as well.

A detailed explanation about the **Kubernetes** networking model can be seen on the Cluster Networking page in the Kubernetes documentation.

If you have experience deploying virtual machines (VMs) based on IaaS solutions, this will sound familiar. The only caveat is that, in **Kubernetes**, the lowest compute unit is not a **container**, but what we call a **pod**.

A pod is a group of co-located containers that share the same IP address. From a networking perspective, a pod can be seen as a virtual machine of physical hosts. The network needs to assign IP addresses to pods, and needs to provide traffic routes between all pods on any nodes.

The three main networking challenges to solve in a container orchestration system are:

- Coupled container-to-container communications (solved by the pod concept).
- Pod-to-pod communications.
- External-to-pod communications (solved by the services concept, which we will discuss later).

Kubernetes expects the network configuration to enable pod-to-pod communications to be available; it will not do it for you.

Tim Hockin, one of the lead **Kubernetes** developers, has created a very useful slide deck to understand the **Kubernetes** networking *An Illustrated Guide to Kubernetes Networking*.