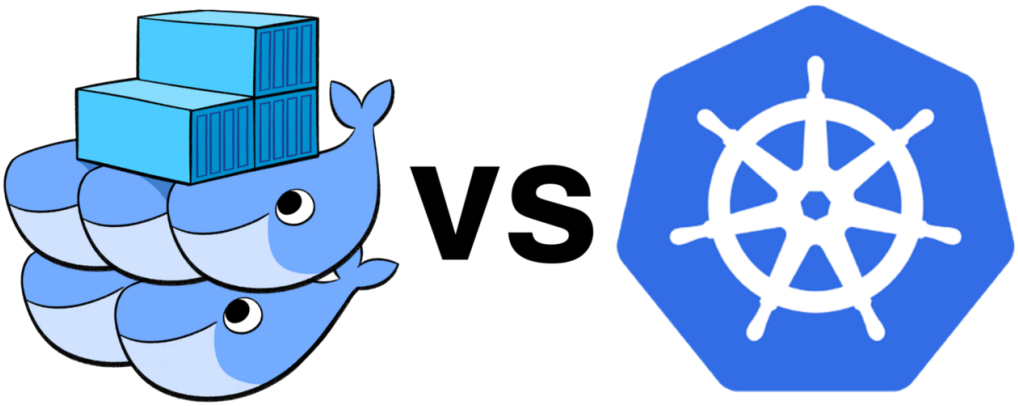



# Docker and Kubernetes for Layman

Published on October 5, 2018

**Aayush Shrut**
Consultant at Deloitte

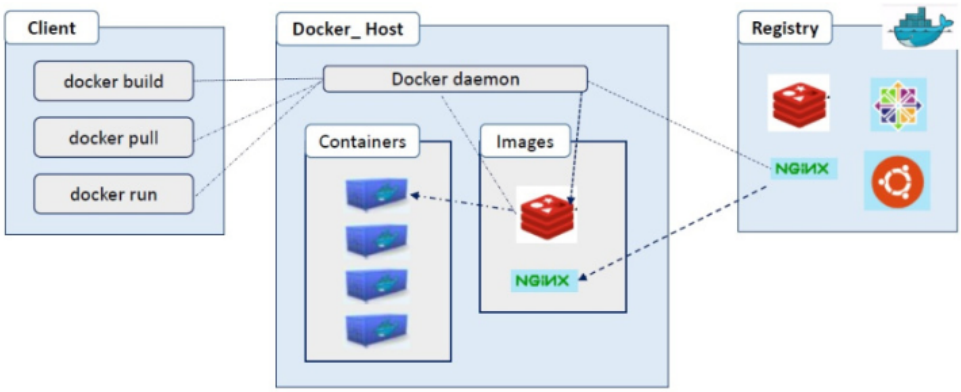


In our previous article, we explored the container ecosystem in detail. We learned how Containers work, its basic architecture, and how they are an evolution to VMs. In this article, we explore two of the most popular software in container ecosystem, knowledge of which can land you a pretty good job (hope this gets your attention!). We introduce, **Docker: The container solution and Kubernetes: The container orchestration solution.**

## What is Docker?

Docker is a computer program that offers "*operating system level virtualization*" using containernerd container runtime library. If you recall from previous article, the runtime library is the engine which enables containerization. Docker is simply the most well accepted (and simple!) container platform. Other alternatives such as rkt, LXD and even Windows Containers (in Server 2016) also exist.

Docker architecture follows a **client-server** model, and contains three distinct layers:

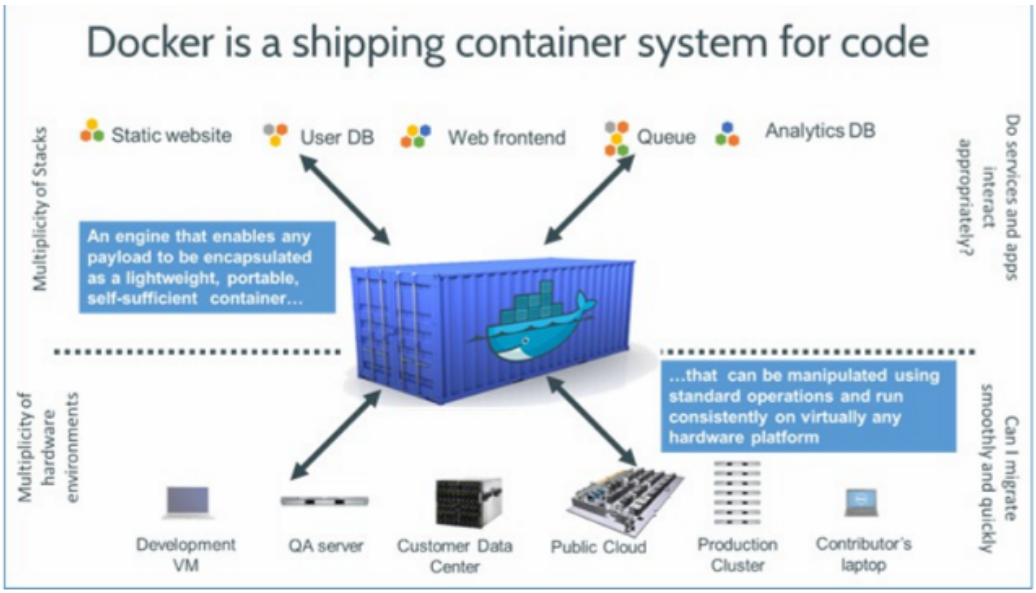



1. **Docker Client**: The UI of Docker, interacts with Docker daemon (using REST API) to execute commands. It need not necessary run on host computer.

2. **Docker Daemon (dockerd):** Running only on the host computer, it executes the client's commands. It does all the heavy work of building, running and maintaining *container objects* such as networks, volumes, images, containers etc. (read the next section).

3. **Docker Registry:** The central stateless server storing your docker images. It can either be your local server, or the commonly used (and free!) Docker hub.

### A quick 101 on containers vs container images:




A Docker Container

In the container world, the above box (containing our application and all its dependencies) is referred to as an image. A running instance of this box (container image) is referred to as a container. **We can spin multiple containers from the same image.** If you are familiar with VM terminologies, think of image as the base VM image, and containers as snapshots. Or in object oriented analogy, *think of images as class and containers as objects.*

An image contains the application, its dependencies and the user-space libraries. **An image does not contain any kernel-space components.** When a container is created from an image, **it runs as a process on the host's kernel**. It is the host kernel's job to isolate and provide resources to each container.

Recall that containers use Union File System to create a layered file system. Docker images are close representation of that. What it means is that, suppose you want to build a custom image called "Ubuntu + Apache + Django + React + fancyFramework running your python application. Normally, you will install Ubuntu, and install all the frameworks and dependencies manually (PIPing away time in the process).

With the layered UFS approach of Docker, all you need to do is specify your requirements in a configuration file called **Dockerfile,** which will automatically install everything **on top of Ubuntu BASE image**. Each and every installation will **add a layer** on top of Ubuntu base image, creating your docker image.

The beauty of this is that, suppose in the future, you get a fancyFancierframework. So instead of good old days of repeating the steps all over again (Ubuntu VM + installation), you can directly install your framework on top of your existing image! And voila, your fancyFancierframework is ready and deployed (*added as a layer*). Another advantage of this is that, suppose you discover a vulnerability in Ubuntu 14.04 image. Now, since all other layers are **inheriting their attributes** from this base layer of Ubuntu, you can specify a new base image and all the attributes will inherit the properties. You can even create branches, remove some frameworks, add new ones, and do tons of other things. Simply put, your software and frameworks are *GITified* (if there exists such term i.e).

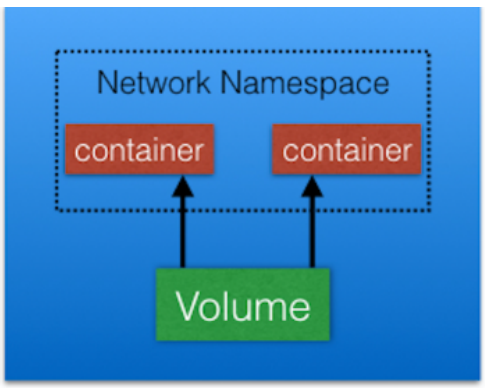
So far, we have discussed containers as a single isolat

Suppose, we have thousands and thousands of containers deployed (example in Micro Service architecture). How do we maintain such deployments? Additionally, how do we specify that certain services run on certain containers grouped together for application coherency (example containers sharing network stacks, storage etc.). To solve these issues, Google, and now Linux Foundation, gives us, Kubernetes.

### What is Kubernetes?

Kubernetes is a container orchestration software for "*automating deployment, scaling and management of containerized applications*". It can work with container softwares such as Docker.

Kubernetes achieves our previously mentioned use case (multiple containers on different systems working coherently) by something called "***pods***". Pods are essentially **a group of containers sharing the same network namespace and filesystem, running on a "*node*"**. Nodes are nothing but the actual server or host on which pods (containing containers) are running. So for example, if you need 5 servers with a load balancer, you can create a pod which has 5 backend container images + load balancer container, all working together and having a single IP, and running on a server node. We can even deploy a single container, but it must be deployed within a pod. Think of pods as a running VM image, which has multiple applications running simultaneously.




Pods

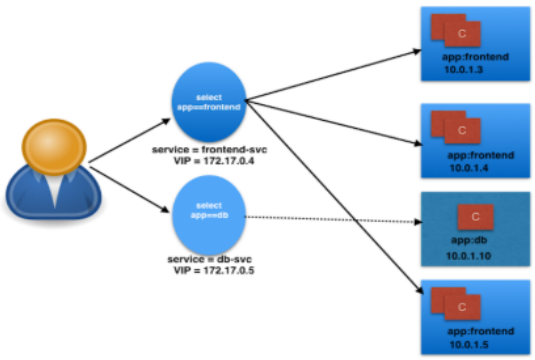
An important concept about pods is that **they are ephemeral in nature**. Hence, they can be destroyed easily (unlike VMs), and need to be re created. Now this creates a problem. Suppose, we want to connect to a certain pod, and it goes down. How can we ensure that pods as a whole can be accessed continuously?

To solve this problem, Kubernetes provides a higher-level abstraction called ***Service***, which *logically groups Pods and provides a policy to access them*. This grouping is achieved via ***Labels*** and ***Selectors*****.**

For example, if our pods are running apps, we can apply labels of app on them, and then group together 3 pods as frontend, and another single pod as database using selector app=frontend and app=db respectively. The resulting logical grouping (called **Service Names**) can be called frontend-svc and db-svc respectively. The ip address to them will be persistent, and hence, users can access the internal pods using these services.

```
kind: Service
apiVersion: v1
metadata:
  name: frontend-svc
spec:
  selector:
    app: frontend
  ports:
    - protocol: TCP
      port: 80
      targetPort: 5000
```

Additionally, these services provide **automatic load balancing** to inside pods. Hence, the traffic to your 3 frontend pods will be automatically balanced (33% on each pod). Additionally, you can access specific pods using the IP address and port combo, called ***service endpoints***. This **target port for service and receiving ports for pod access is provided when we define services** (see above diagram). So for our example, we would accept request for services on port 80, and forward these requests on port 5000. The ip address of services itself are provided by Kubernetes network (*clusterIP provided by service network*).




Grouping of Pods using the Service object

In the above example, frontend-svc has 3 endpoints: 10.0.1.3:5000, 10.0.1.4:5000, and 10.0.1.5:5000.

If too much details are boggling your mind, think of services as another logical grouping of pods. So we have grouping of containers as pods, and grouping of pods as services, which are further categorized using labels and selectors. In layman terms, if potato chips are pods, the rack where they are kept are services, pringles are labels, and the potato crisp pizza flavors are selectors.

Together, **pods, services, labels and selectors form the building blocks of Kubernetes cluster.**

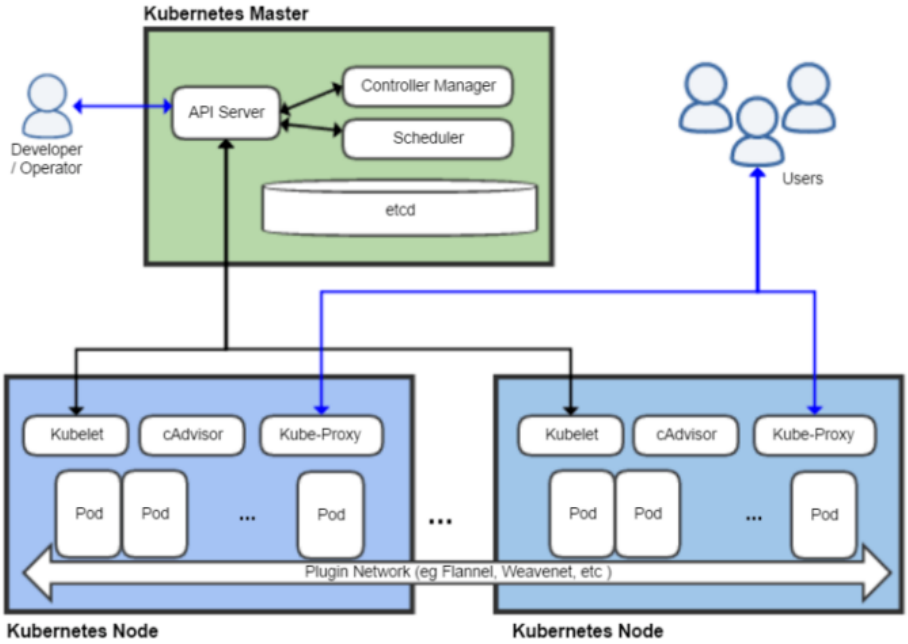
### Architecture:

Kubernetes consists of 3 main components:

1. One or more ***master nodes***: responsible for administrative tasks such as creation of pods. Can be more then one (in high availability mode) for fault tolerance. So if one goes down, another one takes its place. But only one operates at a time.

2. One or more ***worker nodes:*** The actual machine (VM, server, your laptop etc.), where pods are scheduled. Think of it as the computer where your pods (containing different containers) run.

3. ***Distributed key value*** store like etcd: The hash-key set which follows Raft-Consensus algorithm, it essentially stores the configuration details (subnets, secret keys etc.) and cluster state information (which pods are up, master, slave etc.). It can be part of the master node, or, it can be configured externally, in which case, master nodes would connect to it.




**Master node components:**

**API Server**: Primary used for **administrative tasks** (updating, deleting pods etc.). A REST command (from client/operators) is validated and processed, and the resulting state of the cluster is stored in the distributed key-value store.

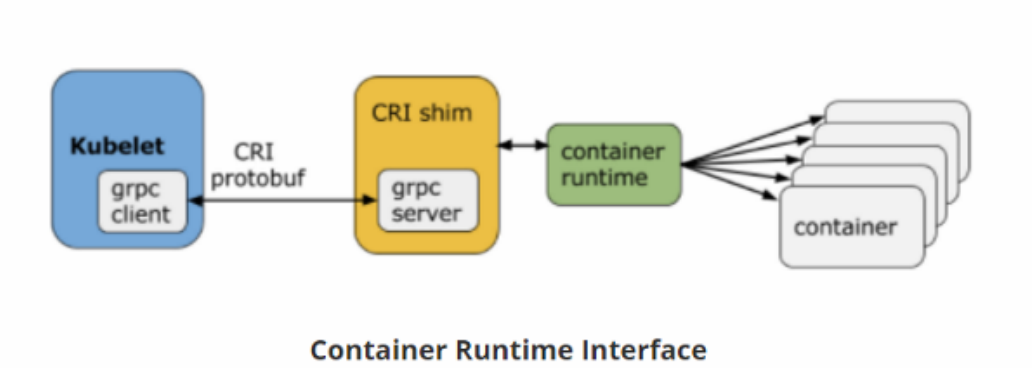
**Scheduler**: It literally schedules the processes on different worker nodes. It has all the information regarding the nodes such as resource usage, user constraints (eg: disk == ssd only), qos parameters etc. It schedules the work in terms of Pods and Services. Hence, **pods are also the scheduling unit of Kubernetes** (for example: memory scheduling, cpu scheduling, similarly, pod scheduling).

**Controller:** It is used to **regulate the state of the Kubernetes cluster**. It achieves this by running something called "*non terminating control loops*". In a control loop, Controller continuously monitors the the current state of the objects using API server. If it does not meet the desired state (knowledge of which is known beforehand), then the **control loop takes corrective steps to make sure that the current state is the same as the desired state**. So for example, a *replication controller* will continuously monitor for desired state (2 replicas), and if current state doesn't meet the desired state (one replica goes down), it automatically create extra pods to meet the desired state.

```
for {
  desired := getDesiredState()
  current := getCurrentState()
  if desired != current {
  makeChanges(desired, current) }
}
```

**Worker node components:**

**Kubelet**: It primarily acts as an **agent to connect with master node**. It is responsible for creating pods based on definitions received through master, and check for the healthy status of the created ones. It connects to the container runtime (like Docker) using something called *Container Runtime Interface* (CRI). For Docker, it uses *dockershim* CRI.




**Kube-Proxy:** Literally a network proxy, it continuously listens to API Server for service endpoint creations. When new services are created or deleted, it continuously updates its routing table so that persistent access can be provided to services (unlike pods). It also acts as a proxy to access our applications from external world (internet).

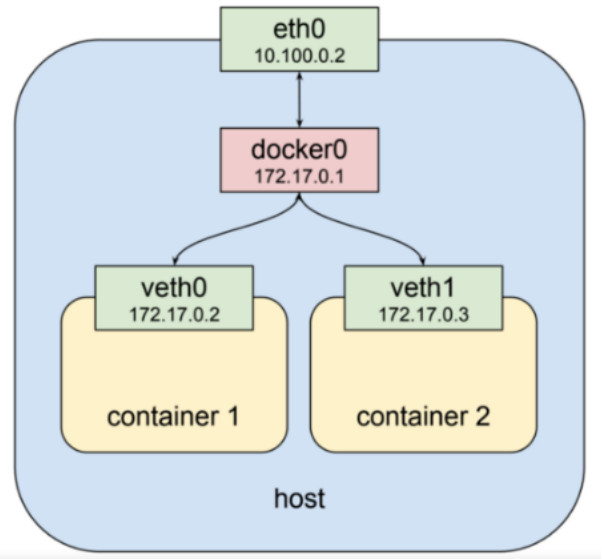
## Networking in Containers:

So far, we have gone through brief overview of architecture of Docker and Kubernetes. Let us now discuss how everything comes into picture by the power of networking!

Before we deep dive into this, a quick refresher of some basic networking concepts should be useful for absoluete beginners in networking. Docker and Kubernetes uses tap devices and linux bridges, almost akin to how VMs communicate (virtual bridges connected to real interface, tap devices connected to virtual interfaces).

**Docker model of networking:**

When we create a docker container, docker creates something called a "**docker bridge**" (*docker0*). Think of it as a **virtual switch**, that connects your actual Network card interface (*eth0*) with the virtual container interface (*veth0*). Any containers (with their isolated network namespace), can communicate with each other using this virtual bridge, as they all exist on the same network range. However, the condition remains that they must be on the same server or host (as virtual bridges are host specific). This again is very similar to how VMs communicate.

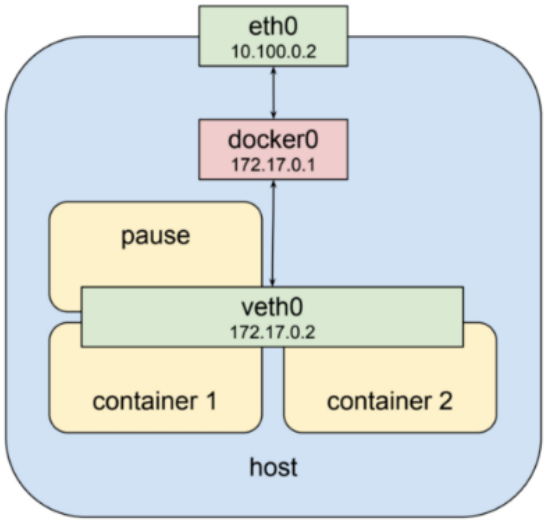



**Kubernetes networking: Container to Container communication inside pods:**

Since in Kubernetes, pods have global IP addresses, we need to implement the above scenario of container communications without using multiple IP addresses. Suppose, two containers share the same IP address. So how can we separately access them? We can try using the power of ports.

We can specify port 80 for container 1 and port 8080 for container 2. This will be similar to how different processes run on your computer, using different ports. The network stack that can be shared can be the friendly "localhost". So we can access container 1 as http://localhost:80/ and container 2 as http://localhost:8080/.

Kubernetes deploys this same IP address and different ports model for their container deployments in the pods. It creates a new container called ***"pause"*** container, whose job is to create a shared network stack (localhost), on which every container inside a pod connects, but uses different ports.

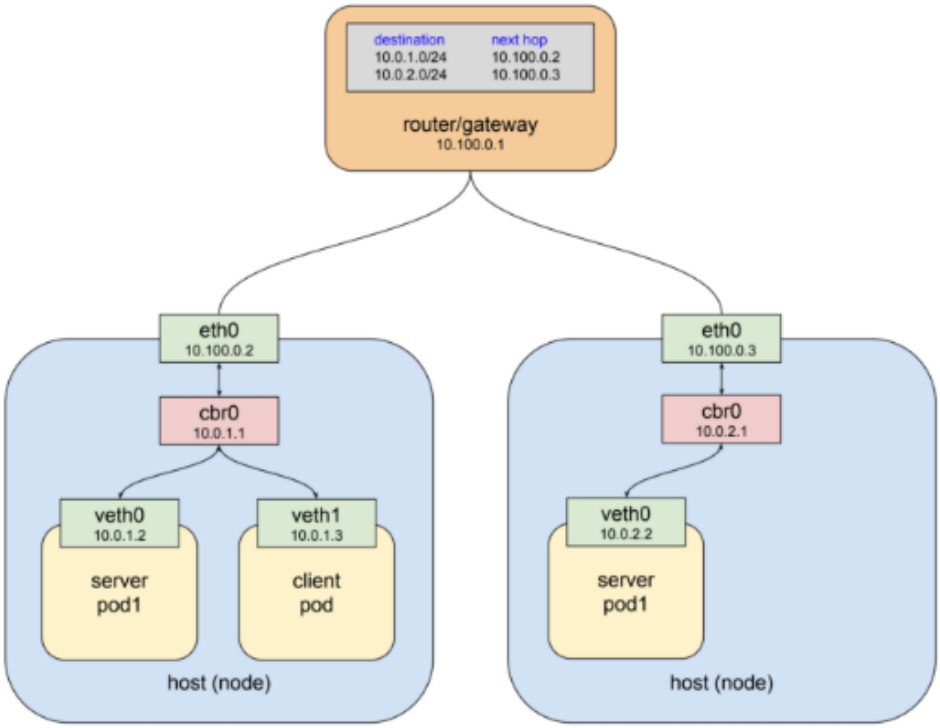



**Kubernetes networking: Pod to Pod communication:**

Since pods can be scheduled on any worker nodes, we might as well discuss the scenario where different pods are running on different nodes.

Any communication between two different networks happens with a router, and within same network happens with a switch. This means that to communicate between 172.17.0.1 to 172.17.0.2, you will use a switch, however, to communicate between 172.17.0.1 to 192.168.53.1, you will use a router.

Applying same concept, node to node communication will use a routing table and a default gateway (provided by Cloud or your server network) to route the packets. The network diagram should be similar to this:




The cbr0 is the same as docker0, a virtual bridge used by the underlying container technology.

One important thing to note is that **there is no NATing between different nodes**, so they exist on the same network. This means that the eth0 interfaces (the real network card) of your nodes are all in the same network. The routing table here actually acts as a switch, since nodes are in the same network.

So how will the IP addresses be provided? It will be done by either the cloud provider (AWS, GCE) or your own server network DNS. What this means that **node IPs are provided by a system external to Kubernetes**. Similiary, container IPs are provided by virtual bridges, using underlying kernel (RFC1918 to be precise). Only pod and service ips are allocated by kubernetes system, which follows "*ip-per-pod*" model, and **groups these containers into localhost namespace and provides a single ip to access them**. Further, it can also **group these pods into services**, and provide ip address from their own internal namespace.

It can happen that your cloud provider, or your own host machine, has limited set of routing rules. Or you might need some additional

management features for node to node communication. This is where, "***overlay network***" comes into picture. The entire scenario mentioned till now can be implemented using an external software called "overlay networks". Examples include Weave, Flannel, Calico etc.

A common misconception is that we absolutely need overlay networks for networking. This is not true. By default, Kubernetes network works fine. In fact, overlay networks add extra overheads. But it is generally an industry standard to use extra software to ease (or disease) the network deployment.

#### Kubernetes networking: External world to Pods communication:

We have read that Kubernetes uses kube-proxy and services for external world communication. However, under the hood, things get really REALLY complicated when we use them. Since they are essential part of Kubernetes network, let's try to provide a very brief overview of how they work.

Let's first recap what we have read, in simple points:

- Containers use localhost (shared namespace) and port combo to communicate. Eg: localhost:80 and localhost:8080. Their IPs are localhost (duh!).
- Pods use virtual bridges (provided by Docker or any other container technology) to communicate with each other. Their IPs are provided by Kubernetes.
- Nodes are actual machines or VMs running the pods. Pods can be deployed on any nodes. Their IP allocations are host user responsibility (using Cloud, or using own DNS etc.).
- Since there is no NATing between nodes, all nodes are kept in the same network.
- The above scenarios can be implemented using overlay network software such as Flannel, Weave, Calico etc.
- Services are logical groupings of pods, whose IP is provided from a different "Service network (clusterIP)". The service routing tables are maintained by kube-proxy running on each worker node, which listens to API server on master node for updates.

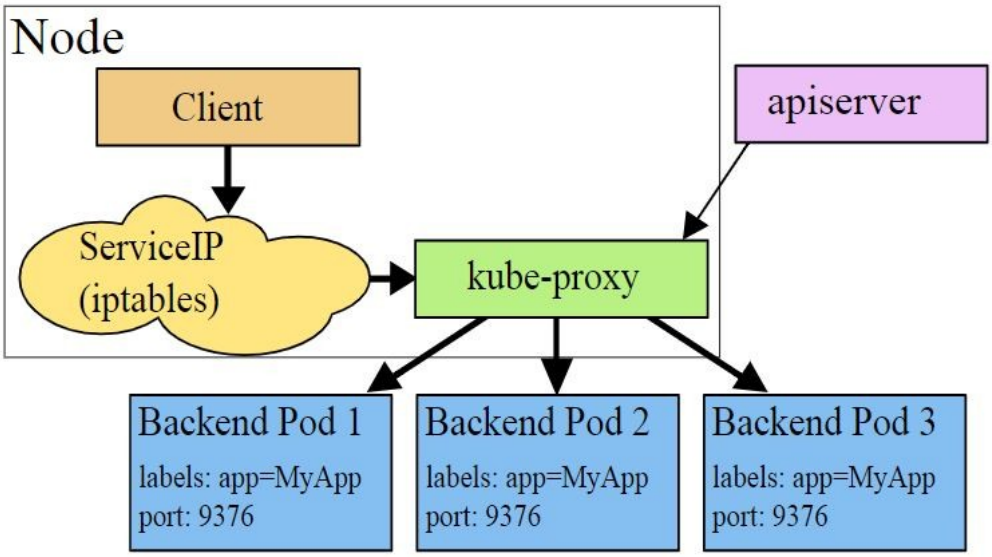
Whews, that's a lot of details on this already long post. Let's focus on one detail. Service IP addresses are very different from pod ips. Service IPs might be 172.x.x.x range, and pod ips in 10.x.x.x range. Now that is not a big problem, but the real magic is that Service IPs don't exist anywhere on the cluster. This means, that no nodes, virtual interfaces, or anything exists on the network, yet, routing is done to services. It is similar to a situation, that you arrive in a LAN party with your laptop containing no network card, and yet you magically play CS (and win it too?).

This magic happens with a kernel level fancy thingy called netfilter and user space fancy implementation called iptables. Details of it is beyond the scope of layman and experts alike, so let's stick to one liners.

**Netfilter**: it is a kernel level proxy, that has full access to the packets' destination and source headers, and can modify the destination (among other things) based on a set of rules. So, it is the big boss, which sees everything, and controls everything.

**IpTables**: Firewall of linux, the rulebook which netfilter uses. (*Pro tip: for any user access issue, run iptables -F. After all, who cares for security when you have arbitrary management deadlines ;) ).*

What happens is that kube-proxy runs on every node's actual physical interface (eth0), and uses the node's Netfilter and IpTables module to route the traffic of services to the correct pods. So the magic of 172.x.x.x converting to 10.x.x.x happens at actual node's physical interface of eth0 by kube-proxy at a kernel level using the big bad boss netfilter and iptables module. Kind of like the diagram given below:




This is the simplest way to describe the process. Under the hood, there are whole lot of other services that aid this.

There are ***NodePort*** services, which are accessible from both pods' real network and services virtual network. Then there are ***LoadBalancer*** services, which give an external IP address to be accessed from external world. There are ***Ingress rules,*** which configures an HTTP (layer 7) load balancer that decouples routing rules to services. Further, there are ***Liveness*** probe, which checks whether the application is alive inside a pod, and ***Readiness*** probe, which checks whether the application is ready to serve traffic or not.

But let us not boggle our mind with such details, and just enjoy the rapid pace of server deployment evolution, from good old days of mainframe, to virtualization, to now present, containerization.

### Conclusion:

This concludes our Docker and Kubernetes for Layman. We started with Docker, on what it is and how it works. We cleared the frequent confusion between images and containers. We then deep dived into Kubernetes, the container orchestration platform. We explored its architecture, and also explored the various networking scenarios under the hood.

Going forward, container orchestration technologies are evolving at a rapid place. Already there are helm charts, which eases software sharing using Kubernetes. Only time will tell the pace

an interesting future awaits.

**Writer's Note**: *If this post helped you in any way possible, do take a minute to like, share or comment :). You can also read some of my previous work here.* As always, *Sharing is Caring!*

## Useful References:

1. Kubernetes Cheat Sheet
2. Docker Cheat Sheet
3. Networking in Kubernetes
4. Installation Guide for Kubernetes
5. Virtual Lab (no signup required!) for Kubernetes

**Mayur Murkya** • 3rd+
Cloud and Redhat Solution Engineer

1y ...

Great blog with useful info.
If possible try to add multimaster (3 master/etcd) in the same which is production grade.

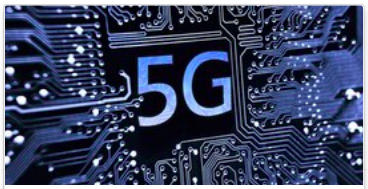
**Aayush Shrut**
Consultant at Deloitte

+ Follow

ɱ Aayush Shrut


**Layman**
ut on LinkedIn

**Containerization for Layman!**
Aayush Shrut on LinkedIn




**5G and Network Slicing for Layman**
Aayush Shrut on LinkedIn

**LTE for Layman (part 3) - The Complete Picture!**
Aayush Shrut on LinkedIn

rticles

Messaging