Docker Containers for Wireless Networks Explained

Abstract:

Containers have many advantages that help enable the delivery of services for wireless operators with 4G and soon to be 5G environments. What a container is and its role in a 4G and 5G wireless environment to improve edge services is explored.

Clint Smith, P.E.
csmith@nextgconnect.com

Containers are making a lot of headway into the wireless telecom space for valid reasons. Containers have many advantages that help enable the delivery of services for wireless operators with 4G and soon to be 5G environments. Containers are not new and have been around for quite some time finding use in the IT domain. They have many unique and important advantages that can help with service delivery in 4G and 5G. Containers are becoming an essential part of the overall edge network solution for 4G,5G and beyond.

So, what is a container.

Containers enable you to separate your applications from the underlying infrastructure making them infrastructure agnostic. Containers utilize virtual environments and there are several different types of containers that can be used. A service provider, enterprise, or utility using containers can deploy, replicate move and even backup processes and workloads faster and more efficiently than when utilizing virtual machines (VM) alone.

One of the more prolific container platforms that utilize a virtualized environment is Docker which is based on the Linux operating system. Containers using Docker run in a virtual environment like NFV however without the need of a hypervisor. Instead of utilizing a hypervisor the container uses the host machines kernel.

Containers have many attributes and some of them are:

- Lightweight: only have what is needed, share the host kernel
- Scalable: using the orchestrator you can automatically replicate containers
- Stackable: you can stack services allowing
- Portable: containers can be run locally or on a cloud.

One of the strengths of containers is that they allow for applications to be distributed making them more efficient. The individual portions of the application in a distributed application are called Services. For example, if you provide an application which is a video game it will likely be made up of many components. The video game application can include a service for an interface, online collaboration, storing application data to a database and so forth. Previously these components of the application would be all in one vertical and be bound together. However, by leveraging the distributed application approach using containers the application can operate more efficiently with less resources being consumed.

Containers in short are individual instances of the application. You can have one image of an application which many instances of that applications use. This allows multiple containers to use the same image at the same time improving its efficiency. Also, every container is isolated from each other because they are able to have their own software, libraries and configuration files to run the application itself. Because containers are created from images the image contains specific content detailing exactly how the container/ application is supposed to function and behave.

So why not just use a virtual machine (VM) instead of a container or use a container in place of a VM. The answer lies in what you are solving.

Both containers and VMs are part of a wireless telecom solution for delivering services because each has unique characteristics which favor specific solutions. Containers as well as VMs are a very essential part of the evolving solution for edge device solutions.

VMs are part of a Network Function Virtualized (NFV) network. An NFV allows an operator to virtualize functions like a Packet Gateway (PGW) or Mobility Management Entity (MME). Depending on the resources an NFV can have several VMs running on the same physical hardware. A VM in short is a virtual service that emulates a hardware server. A VM however relies on the physical hardware that it is running on. The requirements for the VM typically include a separate and specific operating system (OS) as well as hardware dependencies.

An example of a VM would be an MME in 4G LTE. A VM however in this case is a separate instance of the MME itself. The advantage here is that you can scale the amount of VM MMEs based on traffic loads and other requirements through the use of software with the use of the hypervisor. A VM for the other components which make up a 4G and

5G network can be run on the same server or between different servers. The hypervisor in an NFV environment controls what VM is spun up and down depending on the rules or policies it has been instructed to function with. Therefore, with an NFV environment the VM is managed by rules for how they are allocated.

Containers on the other hand while virtual themselves do not control the underling hardware. Instead containers virtualize the OS of the host. Containers can also be setup to perform the same functions as a VM. For instance, a container can be created that has the functionality of a MME. The containerized MME could have all the functionality of an MME or a subset of a full MME depending on what needs to be solved. However, containers are best used for delivering services in a virtual distributed network (vDN). Containers using a vDN can deliver services to edge devices with low latency facilitating mobile edge computing and other requirements.

Both Containers and VM have similar resource isolation and allocation schemes. However, containers virtualize the operating system while VMs virtualize the hardware.

To help show the power of Docker containers for applications a simple diagram is presented in figure 1 showing the difference in the stacks between an NFV and Docker environment.

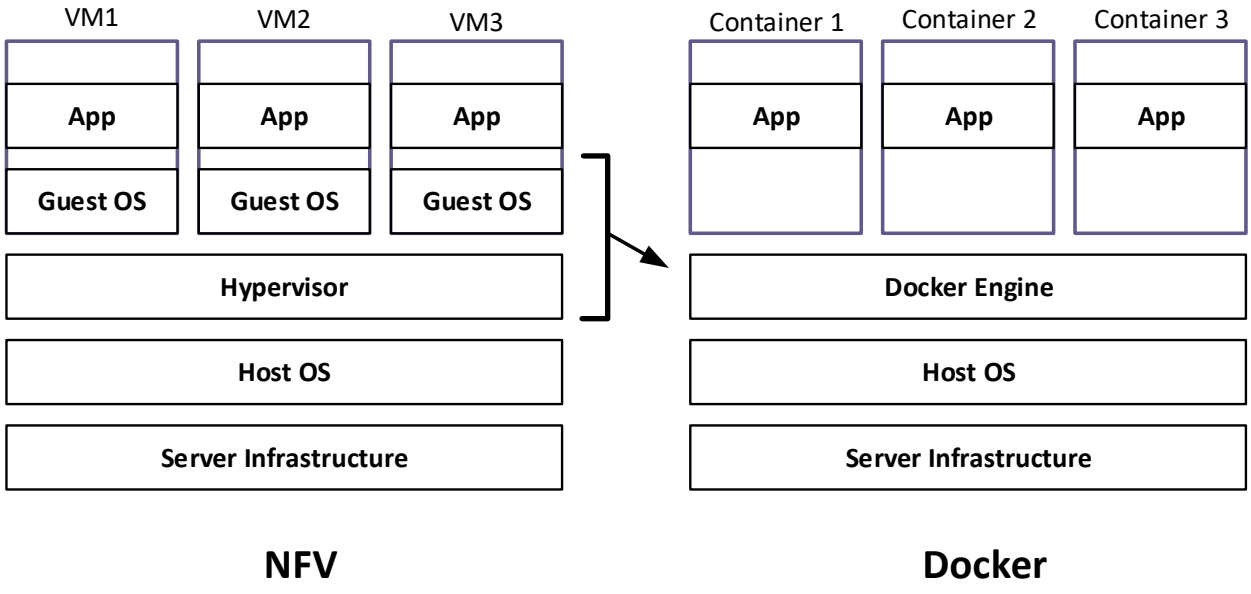

Figure 1: Virtual and Docker comparison

With an NFV and Docker platform there are some common items. Specifically, the server is the entity that is used to host multiple VMs or containers. The Host OS is the OS for the server which is either Linux or Windows. However, NFV and Docker the differences begin after the Host OS layer.

With an NFV platform there are multiple instances of the operating systems. The Guest OS resides on top of a hypervisor in an NFV which is above the host OS. Whereas in Docker the host OS is shared amongst the containers. The applications for the NFV network use the Guest OS and is unique for each application. However, with Docker the application is now a container that can be run using a shared OS.

In addition, the Guest OS in an NFV has a full implementation of the OS that the VM uses. With an NFV the VM could be Linux or Microsoft meaning that every VM that is spun up also has a separate OS consuming valuable resources and time. However, with Docker only the required resources are needed to run the application since the container bundles only the needed libraries for the application to run, since it shares the host OS. Containers can therefore be spun up faster and consume less resources than VMs.

However, you can also run containers within a VM. Figure 2 shows an example of containers within a VM environment. For instance, if you had both Linux and Microsoft applications using the same Host OS it would not work due to kernel incompatibility. Obviously, capacity is sacrificed for flexibility in Figure 2.

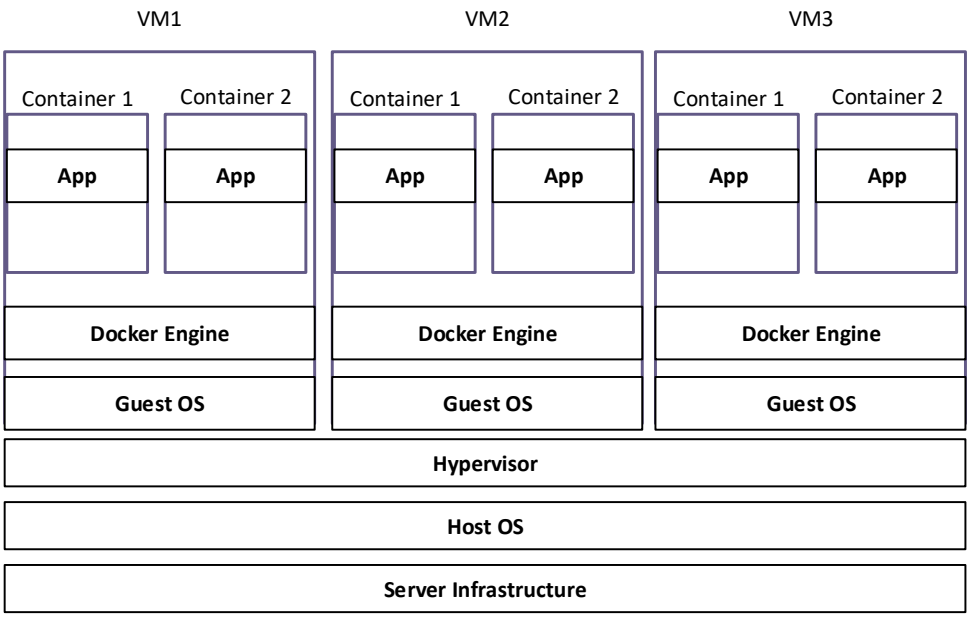

Figure 2: Docker

Therefore, the use of a VM or Container should be considered based on what needs to be solved. Using both containers and VMs in a deployment can be helpful, because they each have their unique attributes which should be leveraged based on the design criteria.

The focus of this article is containers, however, it was necessary to review and attempt to put into context some high-level conceptual points related to virtual environments.

The operating system for containers which is more applicable to wireless networks is Docker. Docker is an open platform that uses containers for creating, delivering and running applications that are very efficient and flexible. Docker, through the use of containers, enables you to separate your applications from infrastructure making it more portable and quicker to spin up when needed.

Containers are considered lightweight because they only need the host machines kernel and not a hypervisor meaning fewer computing resources are needed. Because containers are lightweight many containers can be run with the same computing resources required for a single VM.

Recapping containers virtualize the operating system while VM's virtualize the hardware.

Therefore, the strength of containers is that it provides a common method to run your code and or application making it hardware agnostic and more efficient.

The reason Docker containers (DC) are more efficient for running applications is that it decouples the application from the operating systems. Because a DC can decouple the application from an operating system it enables a more

efficient deployment of the application. In other words, a user utilizing a DC can run their application using only the resources it needs instead of having additional resources assigned that it does not require to function, the bloat.

A DC is designed to be isolated from other DC's by running the application entirely within itself. A fundamental key attribute of a DC is that it shares the kernel of the host operating system (OS). Because the DC shares the kernel the container can be run on a host of environments without the worry of software dependencies because everything needed by the application is contained in the container.

However, the Docker platform performs more functions than just running containers. The Docker platform using GO enables the creation and usage of images, networks, plug ins, and a host of other items. Using the Docker platform wireless operators, enterprises and utilities can easily create applications and ship them in containers that can be deployed anywhere.

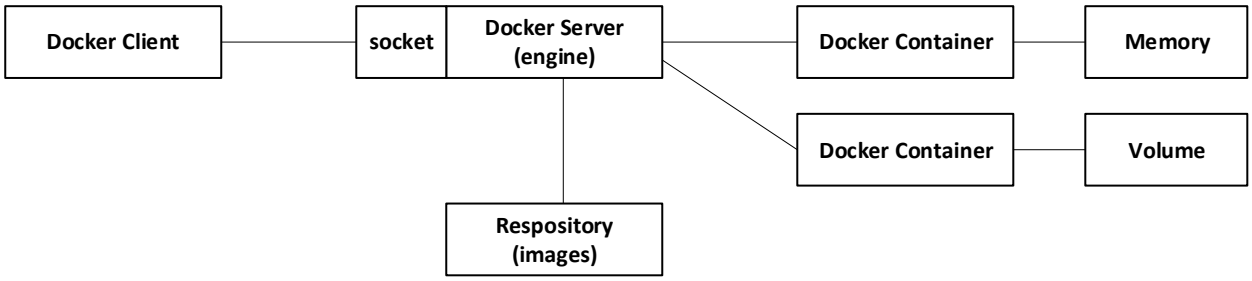

Figure 3: Docker Environment

However, what exactly is a container? Specifically, a container is a run time environment and uses an image which is a package that contains all the items the application needs to function like the config files, code, libraries etc.

Figure 3 representing a Docker environment that shows more than one container along with several other components. Docker has multiple components all with a specific function that they perform.

The main components for Docker are:

- Docker Client
- Docker Server (engine) DE
- Docker Containers (DC)
- Image (stored in public or private repositories)
- Volumes (this holds the data)
- Networking.

Docker uses a client server architecture where the server receives commands from a client over a socket or network interface (in case the server is remote) to run and interact with a container. The Docker client program can be remote, or it can also reside inside Docker as well.

The Docker container is unique in that it does not use more resources, CPU, memory and other resources than it needs. Think of it as running a minimum operating system necessary for the application. What controls the resources used by the container is the kernel.

At the heart of a DC is the kernel and the kernel manages the DC. In essence the same kernel is shared across containers in the same hardware environment.

So, what is a kernel.

A Kernel is a key component for any software platform and application to operate. A kernel does many things however the following are the more relevant attributes:

- Defines the libraries and other attributes of a program/application.
- Controls the underlying hardware by allocating computing resources like memory, CPU, networks and so.
- Starts and schedules programs.
- Controls and organizes storage.
- Facilitates messages being passed between programs/applications.

Kernels not only enable the running of containers in a Docker environment they also restrict the capabilities to just what is needed making the container lightweight, portable and secure. For example, not all the libraries for a particular OS are included with the application in a container. A kernel can also be used to restrict who has access or what is allowed to be changed.

The Docker Engine (DE) is an essential element in the Docker system that enables the ability to deploy and run applications within a container. The DE is a client-servicer application and is called *dockerd* which is a daemon. The DE utilizes CLI REST API for interacting with other Docker elements. The DE components are shown in figure 4 and highlights some of the interactions between the client, host and registry.

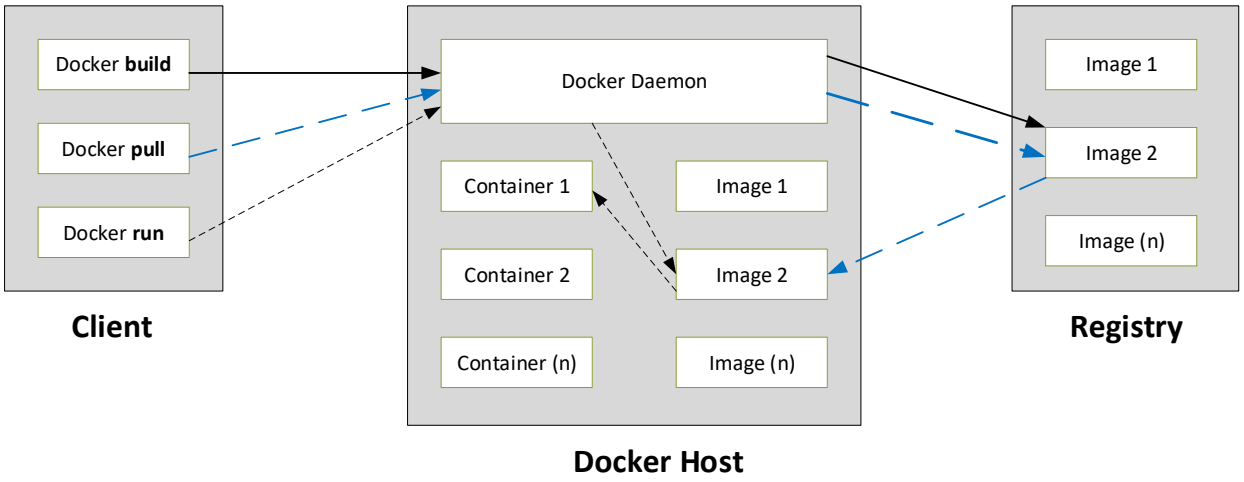

Figure 4: Docker Engine components

In figure 4 three distinct functions are shown: (1) build process flow, (2) pull process, (3) run. Each is represented in Figure 4 by a different line format that shows how it relates to the client, Docker host and registry.

The Docker platform has many components as you would suspect. Referring to figure 4 Docker components can be thought of in three layers: software, objects and registries (SOR). Tools like Swarm and Kubernetes are used to help manage the SOR.

The three SOR layers are:

1. Software: This involves two main software functions
   - Daemon- used by the Docker engine to manage and handle container objects, (network and volumes).
   - Client- uses a CLI function to interact with the Daemon. The client is the primary method for interacting with Docker via Docker API and can communicate with more than one Docker daemon
     a. LCOW- allows you to run Docker containers on Linux

   b. WCOW – allows you to run Docker containers on Windows
   c. MAC - allows you to run Docker containers on MAC
2. Objects: There are three types of objects with Docker: images, containers and services.
   - Image: This is what is used to store and then ship applications. In short, the image is used to build the container and is only readable.
   - Container: This runs the application and is the execution layer for Docker. Containers are images that have a writable layer. Several containers can be linked together for a tiered application.
   - Services: This is a series of containers that function in distributed fashion and is how you scale containers across multiple Docker daemons. You can also define the number of replications allowed for any instance at any given time. The service is load-balanced across all nodes.
3. Registries: This is a repository for Docker images and can be both public and private. Images can be pulled from or pushed to the registry from the Docker client.

A container as mentioned previously is a runtime implementation of an image. Many containers can be started from one image which is a key attribute for service delivery to the edge of the network. Another key attribute is the time it takes to start and stop a container is much shorter than a VM and it consumes less resources.

A container is what is thought about when mentioning Docker however it is only a part. In essence a container is a runnable instance of a particular image. In Docker a container is isolated from other containers and the degree of isolation is determined by how you configure it. Therefore, a container is basically defined by its image and the associated configuration for it.

A container's life cycle is shown in figure 5.

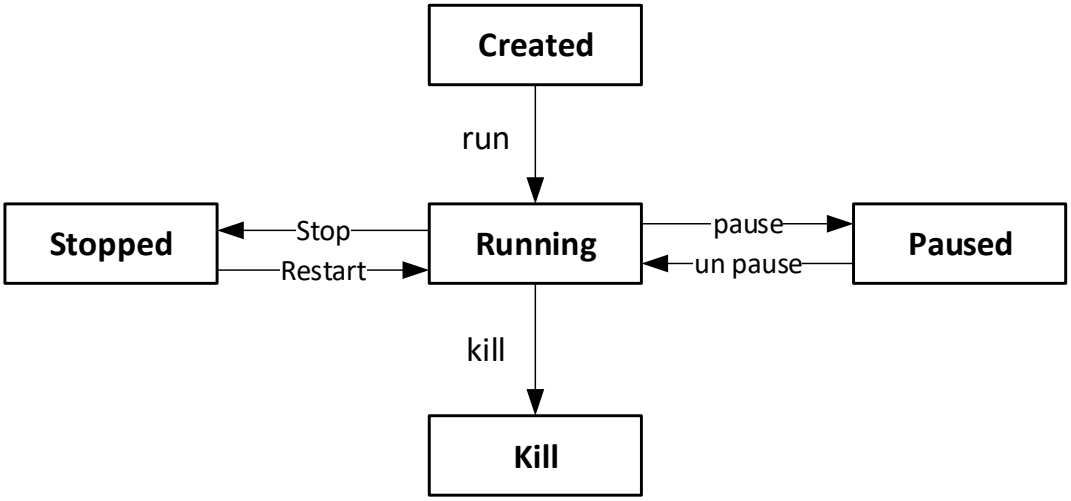

Figure 5: Container Life

Looking at figure 5 a container's life begins when it is created. When the container is created it is really stored in the repository as an image and becomes a container where it is run. During a containers life it can be run, stopped, paused and killed over and over again. However, with all software/applications it to has a life span and changes are needed over time for a host of reasons.

To make a container using an image the docker engine takes the image from the repository then adds a read-write filesystem on top of the image in the container and initializes all the various settings including network ports, container name, ID and resource limits.

Several kernel features that also help enable containers and their functionality includes namespace, control groups and UnionFS.

- Namespaces basically are used by Docker to provide isolated workspaces in Docker which is the container itself. Each instance of the container runs a separate namespace and the containers access is limited to that particular namespace. Using namespaces, a container can have its own unique network interfaces, IP address etc. Therefore, each container will have its own namespace and the processes running inside that namespace will not have any privileges outside its namespace.

- Control Groups also called cgroups, limits the resources an application is allowed to use. cgroups also allow the Docker engine to share hardware resources between containers. For instance, through cgroups you can limit the amount of CPU or memory a particular container can utilize thereby keeping one container from impacting another due to resource consumption.

- Union File System (unionFS) is the file system structure used by a DC. There are multiple types of file systems depending on the application being run like virtual file systems (vfs) and B-Tree file system (btrfs).

DC's utilize copy-on-write (COW) as a method for improving efficiency of file sharing and copying within the application running in the container. Images used to create containers are constructed in layers. COW basically uses the file stored in the writeable layer already there. With COW if a file or directory exists in a lower layer within the image itself and another layer needs read access to it the COW maps it to the existing file for use. However, if the file needs to be modified then the file is then copied into that layer and modified.

Images are used in software all the time and Docker uses them as well. Images are read-only instructions and are used to create a container in Docker. What is interesting is an image can be nested or based on another image. Images are built in Docker using dockerfile. Each layer of the image is created and defined by dockerfile. However, if you need to change or update the image only the layer that needs to be changed is modified. How the image is changed or modified is an advantage Docker has with applications and also makes the applications light-weight.

With Docker there are several ways to modify an image which contains the application. Figure 6 shows the interactions an image can have with regards to changing it. The three primary methods for modifying an image are:

1. Pull from the repository
2. Dockerfile
3. Commit a container that has been modified

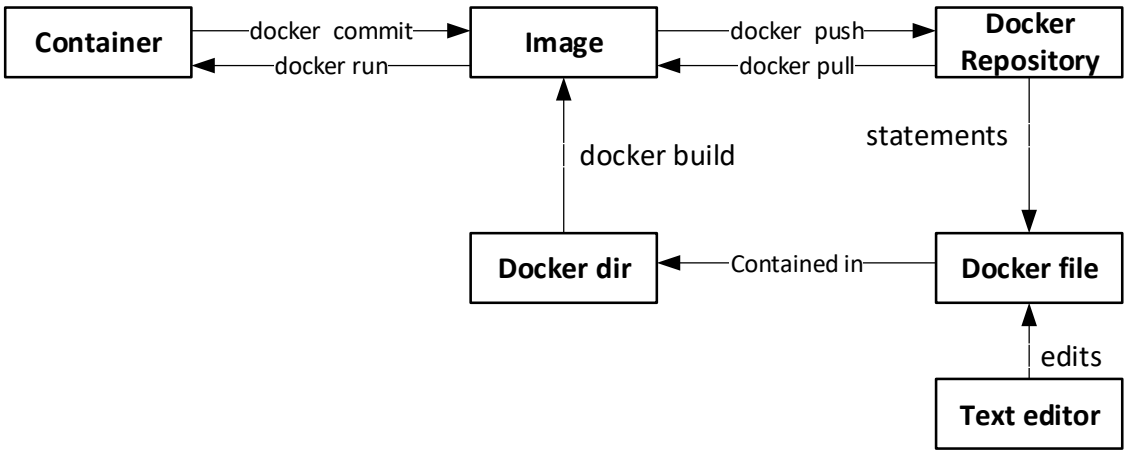

Figure 6: Docker Image

What defines and determines what goes into a container and its environment is determined by dockerfile. A dockerfile is a text file that defines a Docker image and is essentially the manifest for the container. Everything that the application needs to function is defined in the dockerfile. A dockerfile allows the application to be ported to other locations and run identically. Specifically, a dockerfile will define how the application will interface to the outside world, what is needed to run the application, any virtual resources needed to run within the container, network interfaces and external drives and or volumes.

Dockerfiles include the following if you open it with an editor:

- FROM - the portable runtime environment
- WORKDIR - this is the working directory in the container for the application
- COPY – copy the contents into the container
- RUN – install all the needed packages for the app defined in a txt file called requirements.txt
- EXPOSE – this is where you expose a port(s) for the container
- ENV - this is where you name or define the environment variable
- CMD – when the container launches it runs the app defined here, i.e. app.py

Therefore, when the dockerfile is made into an image its made from by copying the contents of the current directory which has both the app defined by CMD and requirements defined in RUN. The output of the app is available from the port defined in EXPOSE.

When creating an image there is also a YAML file with the extension (,yml). The YAML file is not part of the dockerfile but a separate file which is included when building the image itself. The YAML file typically called the docker-compose.yml and provides the instructions for running and scaling the container and services.

The main functions docker-compose.yml instructs the DC to do certain functions include:

- Defining how many replications (replicas) of the image are allowed
- Limiting the type and amount of resources used
- Restart the container if it fails
- Map ports
- Others.

So, what is the difference between an image and container. There are several differences however the one difference that stands out is a container has a writable top layer while an image does not. This means that all data either read or modified is stored in the read-writable layer. Also, when the container is killed, deleted, the data that is in the read-writeable layer is also deleted. However, the image that was used to create the container is unchanged from its initial state regardless of what data was collected.

With applications it's all about the data. Therefore, what you do with the data created depends upon the type of data that is involved, and it is either stateless or stateful.

- Stateless is where you do not save or store any of the data collected or created. If the container restarts everything is lost.
- Stateful is where the application directly reads and writes data to and from a database. When the container restarts or stops the data written is still available.

Docker Volumes are where the data used and or created from the container application is stored. Utilizing volumes is the preferred method for handling data generated and used by DC's. A volume managed by Docker is different than bind mounts since a bind mount is dependent upon the host machines whereas volumes are not dependent on the host machine. Typically, volumes are external to the container.

A volume acts to the application like its file folder with which it can read and write data. This use of volumes is meant to ensure that container is immutable. However, if allowed, a container can also store data it creates in local memory however this data is not persistent.

Figure 7 depicts several different methods for storing or reading data with containers.

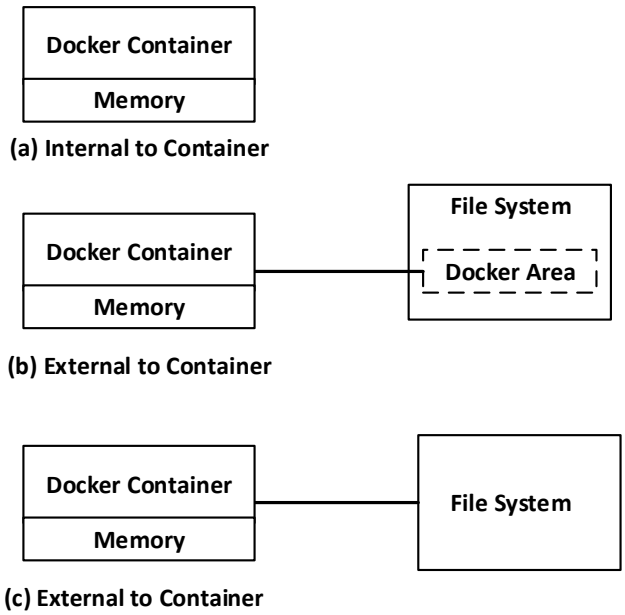

Figure 7: Docker Volumes

The data that the application generates and uses is very important. The ability to read and write data to and from external devices increases the applications functionality and commercial viability.

When a container runs an image, the container has its own writable container layer where all changes or data is stored in that container layer. However multiple containers can and do share access to the same image, but each has their own data state making each unique.

Some of the advantages Volumes have in a Docker environment are:

- Work on both Linux and Windows containers.
- Shared among multiple containers.
- Volumes can be stored on remote hosts or cloud provider.
- Volumes can have their content pre-populated by the container.
- Share volumes between other containers.

There are several ways to create and manage Docker volumes. Each method has its own advantages and disadvantages. The methods to create a Docker volume are:

- Let Docker do it
- Define a specific directory (mount)
- Use Dockerfile

Docker containers need to communicate with the outside world in some fashion. All Docker container communication is done using bridges to create virtual networks. The type of network and interfaces used for the application need to be declared in the dockerfile. However typically the Ethernet bridge in Docker is called docker0 and is created by default.

In order to scale the deployment applications some form of container orchestration is needed. A container orchestration is needed for managing and or scheduling the work of each of the containers to support the applications. In short container orchestration's primary mission is to manage the life cycle of the container which includes provisioning and load balancing.

There are many container orchestrators available however some of the more prevalent ones are Kubernetes, Docker Swarm, and RedHat Openshift.

A container orchestration platform sits on top or rather outside of Docker. The orchestration platform should be able to perform the following general tasks at a minimum.

- Provision and deploy container.
- Provisioning host environment.
- Instantiating containers.
- Allocation of resources between containers.
- Exposing services to entities outside of the cluster
- Load balancing
- Scaling the cluster automatically by adding or removing containers
- Ensuring redundancy and availability of the containers
- Rescheduling/starting failed containers
- Health monitoring of containers and hosts

Both Docker Swarm and Kubernetes perform all the tasks above. However, Docker Inc. has made Kubernetes along with Docker Swarm part of the Docker Community (CE) and Docker Enterprise editions (DE). Kubernetes is more complex than Docker Swarm but with the added benefit of better orchestration.

The following gives a brief overview of both Docker Swarm and Kubernetes.

Docker Swarm (DS) is a Docker-native container orchestration engine that coordinates container placement and management among multiple Docker Engine hosts and is open source. The DS is able to cluster Docker containers called a swarm. DS through the use of clusters enables communication directly with swarm instead of communicating with each DE individually.

Figure 8 is an example of a DS architecture. With DS there are two types of nodes shown in Figure 8 and they are the Manager Nodes and Worker Nodes. With DS a task is a Docker container (DC).

- Manager Node manages the Worker Nodes. The manager node when deploying an application delivers the work or task to the Worker Nodes by distributing and scheduling the tasks.
- Worker Nodes on the other hand run the tasks. The Worker node also reports back to the Manager node informing it of the health and wellbeing of the tasks and containers.

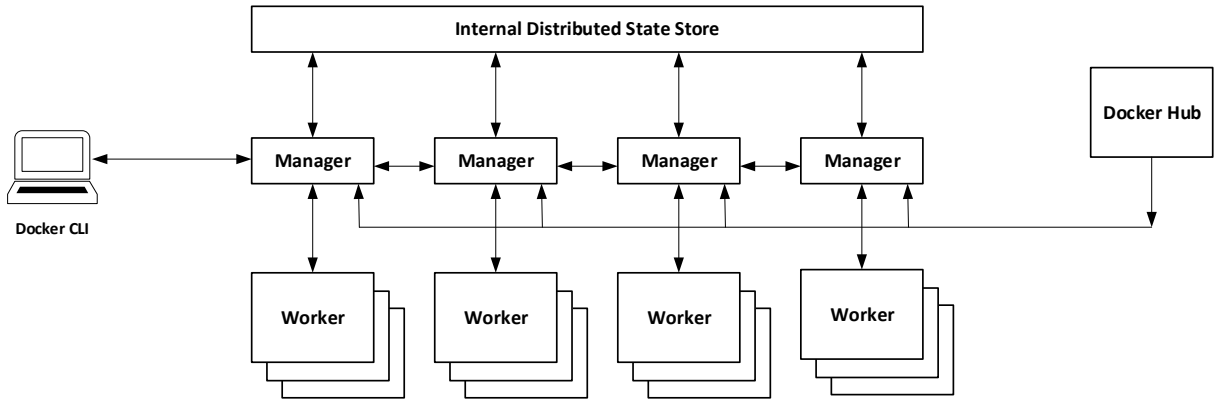

Figure 8 Docker Swarm

Kubernetes however is becoming the orchestration of choice for Docker deployments. Kubernetes is an open-source project. Kubernetes utilizes a client-server architecture. There are several variants to Kubernetes offered by AWS, Google, Azure and others, however they are similar. A high-level diagram is shown for Kubernetes in figure 9.

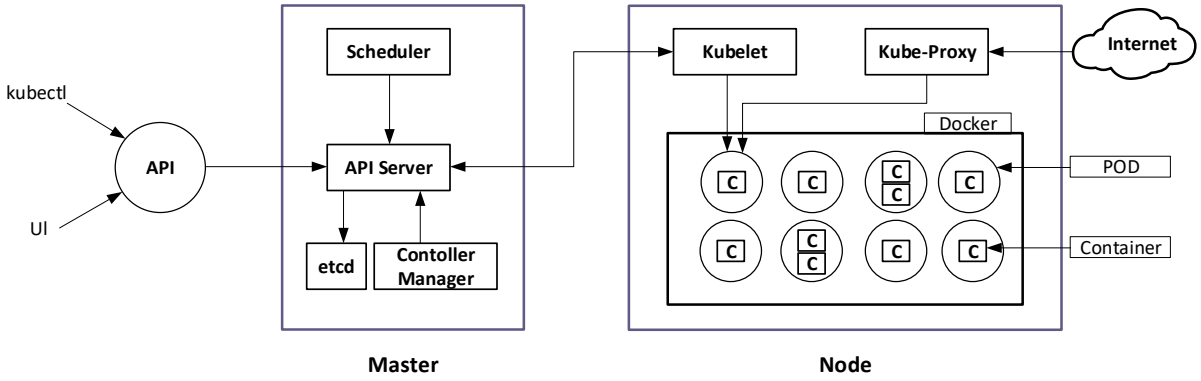

Figure 9: Kubernetes

In figure 9 only one master and node are shown where is reality you can have many nodes associated with a master representing a cluster. There can also be multiple masters for multiple clusters. Therefore, a cluster is a set of nodes with at least one master node and many worker nodes.

The Master manages the schedule and deployment of an application. The Master is able to communicate with the worker and other master nodes through the API server. The main components for a Master include:

- API Server: This facilitates the communication between the various components within Kubernetes.
- Scheduler: It places the workload on the appropriate node and assigns the worker nodes to pods according to how the rules are configured for the assignment.
- Controller Manager: Ensures that the cluster's desired state matches the current state by scaling workloads up and down to match what the configuration calls for.
- etcd: This stores the configuration data which can be accessed by the API Server.

The worker node has the Docker containers and is able to communicate with the repository for retrieving the images. The worker node has many components and the pods hold the containers.

- Kubelet: This is one for each Worker node and is responsible for managing the state of the node including functions like starting and stopping the containers. The Kubelet gets its instructions from the API server in the Master Node.
- Pods: Kubernetes deploys and schedules containers in groups called pods. A pod runs on the same node and shares resources like volumes, kernel namespaces, and an IP address
- Containers: These are Docker containers that reside in a pod.
- Kube-Proxy provides the interface between the worker node and the repository to obtain the image that can be loaded and run in a pod.

However, all this has to start with an ask of some sort. Therefore, kubeclt command or UI is used to instruct the Master Node via a REST API to create a pod. As noted earlier a pod can contain multiple containers.

This article, while somewhat long, is just a brief introduction to containers for use in a virtual environment for wireless operators, enterprises, utilities and public safety. The use of container will facilitate edge computing and service delivery.

An excellent place to continue with containers and Docker is www.docker.com which you can go to and get more detailed information on any of the topics discussed here.

I hope this information has been of some use and I have several more articles that you may also want to read.

Clint Smith, P.E.

Next G Connect
CTO
csmith@nextgconnect.com

Who we are:

NGC is a consulting team of highly skilled and experienced professionals. Our background is in wireless communications for both the commercial and public safety sectors. The team has led deployment and operations spanning decades in the wireless technology. We have designed software and hardware for both network infrastructure and edge devices from concept to POC/FOA. Our current areas of focus include 4G/5G, IoT and security.

The team has collectively been granted over 160 patents in the wireless communication space during their careers. We have also written multiple books used extensively in the industry on wireless technology and published by McGraw-Hill.

Feel free to utilize this information in any presentation or article with the simple request you reference its origin.

If you see something that should be added, changed or simply want to talk about your potential needs please contact us at info@nextgconnect.com or call us at 1.845.987.1787.