What is Docker

Docker is a containerization software. Docker contains all the files for the program to compile and run on various systems, as everything it requires to build and run is within its container. Such as dependencies, as well as specified instructions on how to build and setup the environment for the process to interact within.

Docker uses a client server architecture model. The server is in fact a daemon process running in the background, not tied to a terminal. So when using docker we are in fact using the docker client program to interact with the docker daemon. Thus when running docker, it's important that this daemon is running, for example using systemctl command to start the docker daemon.

Containers are slightly different to virtual machines, though they seem to share a lot of commonality, Rather than superficial virtual hardware using hypervisor, containers use the real hardware and use the kernel of the current machine and resources. The process is just "contained" when running in memory I would presume.

We can manage this containers using control groups. For example you are running three containers in memory, and one suddenly starts eating up all the resources, thus starving the other running containers of resources. We can use control groups in this example to limit the amount of memory and the number of CPU cores etc. the process is allowed. Thus putting limitations on processes to make sure this particular scenario doesn't occur.

Docker containers have a relatively small foot print due to their design and the file system which docker uses for images. They use a filesystem similar to how git works. So we can push and overwrite layers with more layers, or remove layers of text from the image, and then to build the image, we just compile from source, and because it's just layers of source files in the image, it's just char types= 1 byte, thus bytes, the smallest unit, thus source code is more lightweight than a compiled binary, thus we only want to transport the source code, and compile into a binary, when on the target system. The File system used is called UnionFS. Allowing decreased footprint size of the containers and works like code integration tool git, we just push and merge our changes into the base image of the docker container, and the container only needs to know about the past changes etc.

Summary

- Docker has a client server model design
- Linux kernel features used in containers:
  - Namespaces;Cgroups(control groups) and UnionFS

Docker has it's own repository of program images, that we download when creating a container, for example deciding to run a ubuntu container, then If it's not already on my system, then the docker container image will be downloaded to my machine, having everything it needs to co-exist in the native operating system and being self contained and encapsulated somewhat so we can control it, as per the benefit of containers.

Images vs Containers

Images are the templates for making containers. Images just contain text files essentially to build the process. The container is an instance of the image, and has been compiled and is being executed by the CPU in memory.

DockerFile

We Use docker files to write exactly how we want our container environment to be, and what we want to build, how we want to build it,when we want to build it, and configurations. Then we use this dockerFile as an algorithm to follow to build an Image from the instructions in the dockerFile.

Compose

We can also use docker compose files, which use a declarative language called Yaml, Which means it doesn't matter about the sequence of instructions, just declare how you want the image to be built. And is also useful when dealing with multiple containers, Known as a cluster. The root elements of a compose file are:

- Versions
- Services
- Volumes
- Networks

Port mapping

Can use docker to bind a port on the host to a docker container, thus as opposed to just the expose command to expose a port, it doesn't really change anything on the host, Which is beneficial when running web app containers for example or one which requires networking. Using the -p command when running the container we just need to specify the port to bind to. Otherwise rather than specify docker will try and bind a port dynamically.

Networking

docker network command is what allows us to use networking. With networks there are bridge,host,null in docker network groups. We can use the network flag to specify what network we want the container to run on, for example running on the host network or another created network. Bridge allows us to connect to the outside world outside of the container, and the host, whatever ports we open in the container are bounded to the host. Null is no network interface.

Docker network

Persistent storage volumes

We need data storage. And in a container we need a writeable layer. However we still need to deal with persistent storage, which we can do with volumes. For example mounting blockes of memory from the host onto the container, thus when the container ends, changes made are saved to the mounted host data. Thus we use volumes.

Has uses cases such as:

- sharing data among multiple running containers, When the docker host is not guarented to have a given file structure,
- volumes help you decouple the configuration of the docker host from the container run time.
- Useful when you want to store container data on a remote host or cloud provider,rather than locally

- When backing up, migrating containers we can stop the containers from using the volume, and back everything up to the volume prior, and then attach it to the new container.

So we can mount a directory to a container to run and in the container we need to specify where we want to mount this directory on the host. It's similar to mounting a usb to a directory for example.

There is also tmpfs, which creates a memory construct instead, However unlike volumes they are not persistent.

Docker volume
gets added ontop of each instruction, kinda like layers. Which is a useful way of visualizing the filesystem

Container orchestration with docker swarm mode

Docker swarm is useful for situations where container applications reach a certain level of complexity and scale and thus resulting in several machines being required for the container application/s. Container orchestration products and tools allows you to manage multiple hosts in concert. Swarm mode allows for Container orchestration. Swarm mode allowing us to control a cluster of machines.

Features of swarm mode:

- integrated cluster management with the docker engine
- Declarative service model when interacting with docker daemon, sequence of commands is not important
- State monitoring thus desired state reconciliation
- Certificates and crptographic tokens to secure clusters
- Container orchestration features:
  - service scaling
  - multi-host networking
  - resource-aware scheduling
  - load balancing ,rolling updates
  - restart policies

Swarm mode concepts

A swarm is one or more docker engines running in swarm mode. Each instance of the docker engine in the swarm is called a node and it is possible to run multiple nodes on a single machine. Every swarm requires a manager. Managers specifications from users and drive the swarm to the specified desired state through the use of worker nodes. Workers are responsible for running the delegated work. Workers also run agents which report back to the managers on the status of their work. A node can either be a manager or a worker.
A service is made up of user submitted specifications to managers. The service declares its desired state ,including networks and volumes, the number of replicas, resource constraints and other details. There are two kind of services, replicated and global.

In a replicated service the number of replicas for a replicated service is based on the scale desired. Global service allows one unit of work for each node in the swarm and can be useful for monitoring services.

The units of work delegated by managers to realize a service configuration is called a task. Tasks correspond to running containers that are replicas of the service. Managers schedule tasks across the nodes in the swarm.

Swarm networking

We require the overlay networking driver so swarm is natively supported, as the network requirements are specific when clustering, as they need to communicate with each other, the nodes etc.

When working with swarms it's important to have a look over the firewall and isolation rules in the overlay network, and firewalls and network isolation rules also apply to overlay networks as they do bridge networks.

Node influence

We can influence the nodes which run the services by CPU and memory reservations, Placement constraints and placement preferences.

Consistency

Docker swarm in order to create consistency uses Raft consensus. This works be electing a manager node as the leader. The elected leader has the following functions:

- Makes decisions for changing the state of a cluster
- accepts new service requests and service updates and scheduling tasks
- Decisions are acted only where there is a quorum (a committee to vote)

The first manager is automatically the leader, and if the leader fails, it moves to the next manager node in line within the cluster and thus next manager is elected.

More managers result in an increase managerial traffic required for maintaining a consistent view of the cluster and time to achieve consensus. The rules for managers are the following:

- Should have an odd number of managers
- A single manager swarm is acceptable for development and test swarms
- A 3 manager swarm can tolerate 1 failure. A 5 manager swarm can tolerate 2 failures
- Docker recommends a maximum of 7 managers
- Distribute managers across at least 3 availability zones.

Sometimes it's important to decide if we let the managers do work at all? If they are overly utilized it could result in performance issues, because the managers are also used for Raft consensus. Thus having over utilized managers can result in performance issues to the swarm. We can mitigate this by resource reservation thus making sure managers don't become starved of resources. We can also prevent any work being schedule on manager nodes by draining them.

So there are trade-offs to manager nodes, what about worker nodes? There are not really ant disadvantage to more worker nodes. More workers give a greater capacity of productivity and running services and improving service fault tolerance. Furthermore the more workers, doesn't effect the performance of Manager's raft consensus process. Worker nodes also participate in their

own consensus process exchanging overlay network information nodes participate in a weekly highly scalable gossip protocol called SWIM.

Rafts logs are used to share between manager nodes to help with the quorum. And records all the changes a leader makes to a swarm.

SWARM security

cluster management

- Swarm uses PKI to secure swarm communication and state
- Swarm nodes encrypt all control pane communication using mutual TLS
- Docker assigns a manager that automatically creates several resources:
  - root certificate , key pair, worker token, manager token.
- When a new node joins swarm , the manager issues a new certificate which the node uses for communication.
- New managers also get a copy of root certificate, incase they need to take leadership incase of an election.
- We can swap out certificates, and the nodes in the cluster will change accordingly their certificates.

We can also enable encryption overlay networks from the outset. The raft logs are also encrypted at rest on the manager nodes. Swarm secrets are stored in the raft logs, thus it’s important they are encrypted.

When locking a swarm with a key, the key is persistent by default and stored on disk. Swarm allows us to use a key that is never persistent to disk with autolock. When a swarm is autolocked , you must provide a key when starting the docker daemon.