An Introduction to the two Most Popular Container Orchestration Platforms: Docker Swarm & K8s

Nicolai Hellesnes, Diego Leon

March 22, 2020

1 Introduction

The use of containers is increasing and with that comes challenges. It can be trivial to manage a few containers but larger applications can have hundreds of containers spread across multiple hosts. Management suddenly gets much more complicated. Thus container orchestration is essential for scaling. Docker Swarm and K8s are the most popular orchestration platforms today. The purpose is therefore to give the reader a comprehensive introduction to both platforms.

2 Containers

Containers are software that package all the code and dependencies needed for an application. A container allows one or a group of processes to run in a loosely isolated environment. The environment is loosely isolated because it uses the same kernel as the host machine. The container will only be able to see the processes that it is running, while the host machine may be able to see the processes that the container is running.[1]

One big advantage of containers is that it allows applications to run in different environments without needing to change the environment that the container is run in before the application can run. This eliminates the common problem in software development where a piece of software can run on a developer's machine environment but inexplicably does not run on another developer's machine.[2]

The process for creating a container is easy. For example with docker you start with creating a dockerfile, this file states everything that is needed for the container. The file then gets built which results in an image. When this image is executed a container is created. This process makes containers very easy and flexible to work with and makes deployment incredible fast. Under the hood containers basically boils down to clever uses of namespaces and cgroups, which is what makes them incredibly lightweight.[3]

3 Docker Swarm

When using Docker Swarm you have two options, you can use the standalone Docker Swarm, or the Docker Swarm mode which is integrated into the Docker engine in Docker 1.12 and higher. Docker Swarm is seen as the legacy option while Docker swarm mode is the recommended option. This introduction to Docker Swarm will focus on the Docker Swarm mode as the standalone Docker Swarm functionality is seen as a legacy option.[4]

In Docker terminology a swarm is a cluster of Docker engines. Docker swarm mode is the cluster management and orchestration features built into the Docker engine. These features are built into the Docker engine using swarmkit.[5]

There are a few different concepts that have to be understood to understand Docker swarm. Let's start with roles. There are two different roles in a swarm, manager and worker. The managers manage the membership to a swarm and divide work in the cluster. The role of a worker is to execute the work they have been given.

A node is the term for an instance of a Docker engine that is a part of a Docker swarm. One computer can have multiple nodes that are part of a Docker swarm. A node can either perform one of these jobs or both.[6]

In Docker terminology a service is the main structure of a swarm. The service defines what tasks should be executed. When a service is created, you decide what image should be used and which command should be executed.[7]

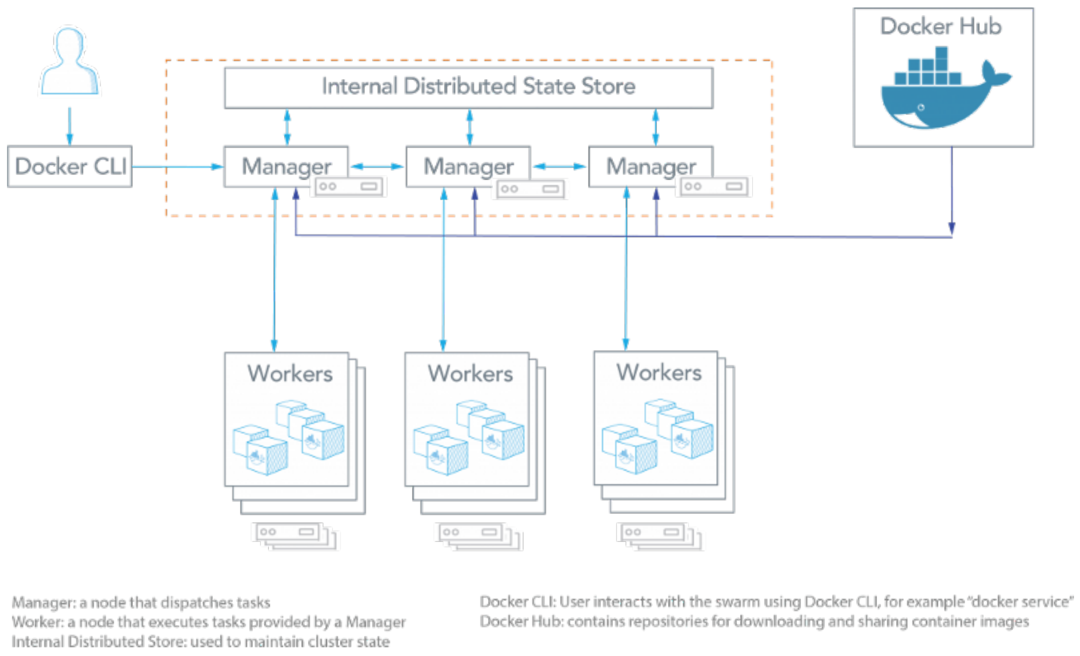

Docker Hub
Internal Distributed State Store
Docker CLI
Manager
Manager
Manager
Workers
Workers
Workers
Manager: a node that dispatches tasks
Worker: a node that executes tasks provided by a Manager
Internal Distributed Store: used to maintain cluster state
Docker CLI: User interacts with the swarm using Docker CLI, for example "docker service"
Docker Hub: contains repositories for downloading and sharing container images

Figure 1: The structure of a Docker Swarm

When you are using a so called replicated service model, the desired number of replicas of a task will be distributed in the swarm. The desired number of replicas would have been needed to be set in the desired state. There also exists something called global services. If a global service is used each available node will each get one task.[8]

You can by hand define the optimal state for a swarm. The optimal state can contain things like the number of replicas you want, and the network resources that should be available. Docker will always try to be in this optimal state. This means that if a node becomes unavailable, Docker will try to schedule the task of the node onto another node.[9]

A container that is running and is part of a swarm service is called a task. The swarm mode uses a decentralized design, this means that the handling of roles are done at runtime and not at deployment time. It is thus possible to build a swarm by using just a single disk image. When it becomes time to deploy an application with a swarm, a service definition should be delivered to a manager node, that manager node will divide tasks to the worker nodes.[10]

A Docker host is a physical or virtual machine that runs Docker. For a swarm to exist there needs to be multiple Docker hosts that are executed in swarm mode. These hosts take the roles as manager and divide up the workload.[11]

To scale a docker swarm the Docker CLI can be used. With the Docker CLI the number of containers for a service can be adjusted. This makes scaling a Docker swarm very easy and flexible.[12]

Load balancing is an important part for scaling a service. With load balancing a service can divide the load over different parts of the service to be able to handle the current load without needing to add additional nodes. In a Docker swarm a swarm manager makes use of ingress load balancing. This ensures high availability even during high loads. A Docker swarm also has an internal DNS component, this component can automatically assign the services in the swarm a DNS entry. The Docker swarm manager makes use of internal load balancing to divide the requests among the different services of the cluster.[13]

4 K8s

Kubernetes is an open-source container orchestration platform originally designed by Google, and is now maintained by the Cloud Native Computing Foundation. It's used for automation, scaling, and management [14]. It gives engineers a centralized system for managing containers.

It can be trivial to manage a few containers but larger applications can have hundreds of containers spread across multiple hosts [15]. If this were to be done manually, an entire team would be dedicated to this.

Dealing with containers include some important tasks such as: balancing the load, connecting them to the outside world, and managing them. A tool is needed to integrate and orchestrate the containers, make them fault tolerant, scale up and down depending on the circumstances, and provide communication across a cluster. A container orchestration tool like Kubernetes is therefore needed.

A K8s cluster is a central component and consists of several machines that could either serve as a master or as a worker node. The K8s master (the control plane) has an important role and serves as an access point which enables interactions between the admin (or other users) and the cluster, with the purpose of scheduling and deploying containers [16][17].

The idea behind Kubernetes is to enforce the "Desired state management". Which means that we are going to feed the master node a specific configuration and it will be up to the API to go out and run that configuration. It does this by informing all the nodes about where to place the application instances and how many to run.

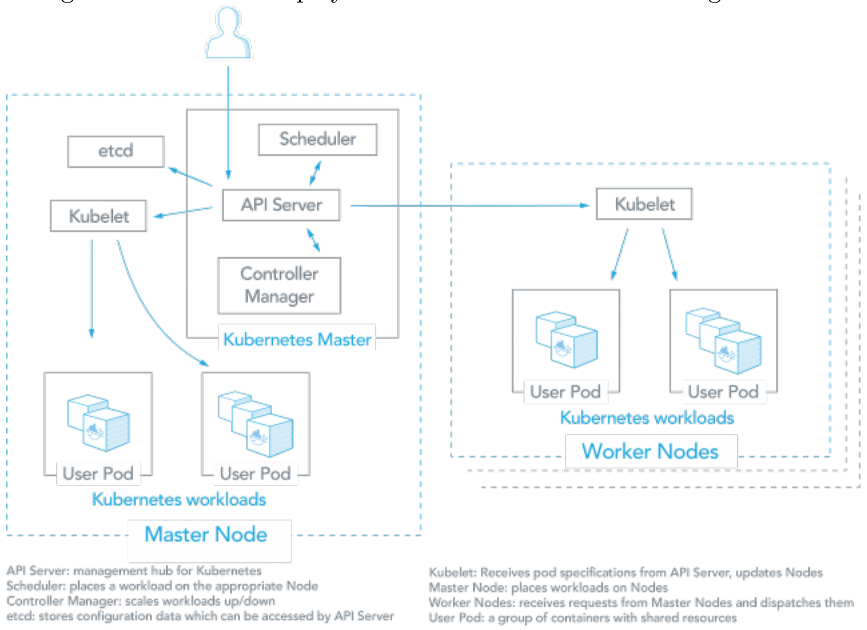
The master node consists of following components:

- Etcd

- kub-apiserver
- kube-controller-manager
- cloud-controller-manager
- kube-scheduler

The ectd is a persistent key-value data store where the master stores the state and configuration data for the whole cluster. All the nodes in the K8s cluster have access to the ectd and use it to manage the containers they are currently running.

The kube-apiserver is said to be the front end for the K8s API server and is utilized by the master to communicate with the rest of the cluster. The kube-apiserver is among other things, responsible for making sure that the configurations of the deployed containers match the configurations in etcd.



API Server: management hub for Kubernetes
Scheduler: places a workload on the appropriate Node
Controller Manager: scales workloads up/down
etcd: stores configuration data which can be accessed by API Server

Kubelet: Receives pod specifications from API Server, updates Nodes
Master Node: places workloads on Nodes
Worker Nodes: receives requests from Master Nodes and dispatches them
User Pod: a group of containers with shared resources

Figure 2: The structure of K8s

The Controller manager is responsible for handling the K8s control loops. These loops manage the state of the cluster through the API server. This service is important and takes care of replicas, node management, and deployments. Part of the service includes monitoring the health of nodes.

The kube scheduler receding inside the master worker is responsible for tracking the node workloads in the cluster and giving work to each node depending on its ability. It looks for new pods that don't have any nodes and assigns them a node based on the requirements.

A container is deployed to all nodes in the kubernetes cluster. The application runs inside the containers. This is managed by a container runtime (such as Docker) which all the nodes in the cluster are configured with.

All K8s nodes have a process called kubelet. This process is responsible for handling the state of the node, which is set based on instructions from the control plane. Some of the states are: stopping, starting, and maintaining containers. It saves the health and performance from the container, node, and pods. These kubelets gather data from the node, pods and containers it shares the node with, and sends it to the control plane which makes scheduling decisions based on this information.

A pod is a scheduling unit that contains one or more containers, and information about how to run them. These containers are guaranteed to be co-located on the host machine[18]. The so-called, desired state, is described in a pod through a K8s object called yaml. These objects are handled by the API server. To define the pods and the number of container instances (replicas) for each pod, a YAML object called "deployment" has to be created.

5 Conclusion

Container orchestration platforms are critical when operating on scale. Selecting the right one for your needs can be hugely important. Both of the platforms have their pros and cons. Docker Swarm has a very simple installation while kubernetes is much more complex to install. Kubernetes comes with a built-in gui while with Docker swarm you have to rely on a third party gui. Both of the platforms are highly scalable. In Kubernetes scaling is accomplished by changing the number of replicas in a Deployment, while with Docker swarm scaling can be accomplished with the CLI.

If you are using Docker to containerize your application, both are viable options as both Docker swarm and Kubernetes work with Docker containers. It can be trivial to manage a few containers but larger applications can have hundreds of containers spread across multiple hosts. Both Kubernetes and Docker Swarm can suit you well in this case. Otherwise if this were to be done manually, an entire team would be dedicated to this.

One advantage of Kubernetes is the large open-source community that it has. Another advantage of Kubernetes is that it offers more functionality than Docker swarm. On the other hand Docker swarm is much more lightweight than Kubernetes while also being much easier to deploy.[21] The complexity of setting up Kubernetes is in most cases not an issue when it comes to cloud deployment. Nowadays, most providers have offers that take away a big portion of the required setup[19].

Docker swarm being built into Docker offers a huge advantage. Docker swarms might be the best choice for individuals or small developer teams that do not deploy often, while Kubernetes is a great choice for all but the smallest and simplest of workloads[20].

References

[1] https://www.ibm.com/cloud/learn/containers

[2] https://cloud.google.com/containers

[3] https://jvns.ca/blog/2016/10/10/what-even-is-a-container/

[4] https://docs.docker.com/swarm/overview/

[5] https://github.com/docker/swarmkit/

[6] https://docs.docker.com/engine/swarm/key-concepts/

[7] https://docs.docker.com/engine/swarm/key-concepts/services-and-tasks

[8] https://docs.docker.com/engine/swarm/services/replicated-or-global-services

[9] https://docs.docker.com/engine/swarm/key-concepts/

[10] https://docs.docker.com/engine/swarm/

[11] https://codefresh.io/docker-tutorial/docker-machine-basics

[12] https://docs.docker.com/engine/swarm/swarm-tutorial/scale-service/

[13] https://docs.docker.com/engine/swarm/key-concepts/load-balancing

[14] https://kubernetes.io/

[15] https://www.infoworld.com/article/3173266/4-reasons-you-should-use-kubernetes.html

[16] https://blog.newrelic.com/engineering/what-is-kubernetes/

[17] https://www.linode.com/docs/kubernetes/beginners-guide-to-kubernetes-part-2-master-nodes-control-plane/

[18] https://kubernetes.io/docs/concepts/workloads/pods/pod/

[19] https://stackify.com/docker-swarm-vs-kubernetes-a-helpful-guide-for-picking-one/

[20] https://stackify.com/docker-swarm-vs-kubernetes-a-helpful-guide-for-picking-one

[21] [https://www.ibm.com/cloud/blog/new-builders/docker-swarm-vs-kubernetes-a-comparison

Figures

Figure 1: https://hackernoon.com/hn-images/1*f8alJuUlWF23Kg-K0teV1A.png

Figure 2: https://hackernoon.com/hn-images/1*uRjGvezQpIZZKKnQ8-7xig.png