RANCHER LABS

July 2017

Comparing Orchestration Engines in Rancher

A closer look at Docker Swarm and Kubernetes

CONTENTS

A key decision faced by DevOps managers is the choice of orchestration solutions for Docker workloads

INTRODUCTION

While Docker has emerged as a defacto standard for containers, users have multiple choices when it comes to deploying and managing containerized application environments. There are a variety of open-source and commercial solutions for on-premises, cloud and hybrid cloud deployments offering different capabilities and feature sets.

A key decision faced by DevOps managers is the choice of an orchestration solution for Docker workloads. Orchestration solutions are evolving at a rapid clip. In this paper, we take detailed look at two popular container orchestration engines: Docker Native Orchestration (aka Swarm) and Kubernetes. Both are good choices depending on an organization's goals, the environment, and types of applications being deployed. While we attempt to look at each orchestration framework objectively, in places it's helpful to compare how a feature is implemented in one framework vs. another, and how implementation details vary.

After reviewing each framework, we provide a short comparative summary, discuss how each orchestration solution is implemented in Rancher, and offer some guidance around what orchestration solution may be appropriate when.

DOCKER NATIVE ORCHESTRATION

OVERVIEW

Docker continues to evolve rapidly. In July of 2016, a year ago at the time of this writing, Docker Engine 1.12 shipped, supporting a native Swarm mode in Docker Engine. Prior to this, Docker users were required to install, configure and manage Swarm clusters separately. The previous stand-alone mode is still supported in Docker, but most users will want to take advantage of the new native Swarm mode. In this paper, we refer to Docker Native Orchestration and Swarm interchangeably, recognizing that Swarm is now a feature of Docker.

To get a sense of how quickly Docker orchestration is evolving, it's useful to look at some of the new capabilities introduced in just the last year:

- **July 2016** – *Docker v1.12* – Swarm was embedded in Docker Engine, and new API objects (services and nodes) were introduced to enable multi-host and multi-container orchestration.
- **January 2017** – *Docker v1.13* – Additional features and bug fixes were added including:
  - Compose file support for the *docker stack deploy* command allowing deployment of multi-host, multi-service stacks using either *DAB* (Distributed Application Bundle) or compose format files.
  - Features related to services including specifying target numbers of instances, rolling update policies and scheduling policy constraints.

Docker Swarm is evolving at a rapid pace with dramatic changes in just the last twelve months

- A re-factored CLI with a more logical command structure and features allowing the new CLI to talk to older service daemons for easier upgrades and administration in mixed environments.
- A variety of new features and commands such as docker service logs, docker system df and docker system prune were introduced to simplify administration.

- **March 2017** – *Docker v17.03* – Docker was restructured into a separate Community Edition (CE) with available *Edge* and *Stable* editions. A new Enterprise Edition (EE) was segmented into three tiers: Basic, Standard and Advanced. Different features are available in different tiers including LDAP/AD integration, role-based access controls, security scanning, and vulnerability monitoring. Release numbers for Docker were also realigned around a more intuitive year / month date format.

There are other changes in Docker as well. Notably, the Docker Hub has been superseded by a new Docker Store, a hosted registry with free and paid images available from various publishers. Docker Cloud is a new service that makes it easy to deploy Docker Swarm environments using resources from various public cloud providers.

ARCHITECTURE

In Docker, a collection of nodes running the Docker Engine can be grouped into a Swarm. Initializing a Swarm is easy (docker swarm init), and administrators can add nodes to the Swarm at any time (docker swarm join) as either masters or workers. There can be multiple master nodes (for high availability) and masters share state via an internal implementation of the RAFT consensus algorithm. An internal distributed data store is used by cluster nodes in Swarm mode, avoiding the need for separate key-value-store. Secrets (a feature of Docker) are encrypted and maintained in the distributed data store as well.

The architecture of a Docker Swarm is pictured in Figure 1. There are relatively few components because the Swarm functionality is built directly into Docker (dockerd) running on each host. As a result, Swarm has few dependencies making installation and configuration relatively simple.

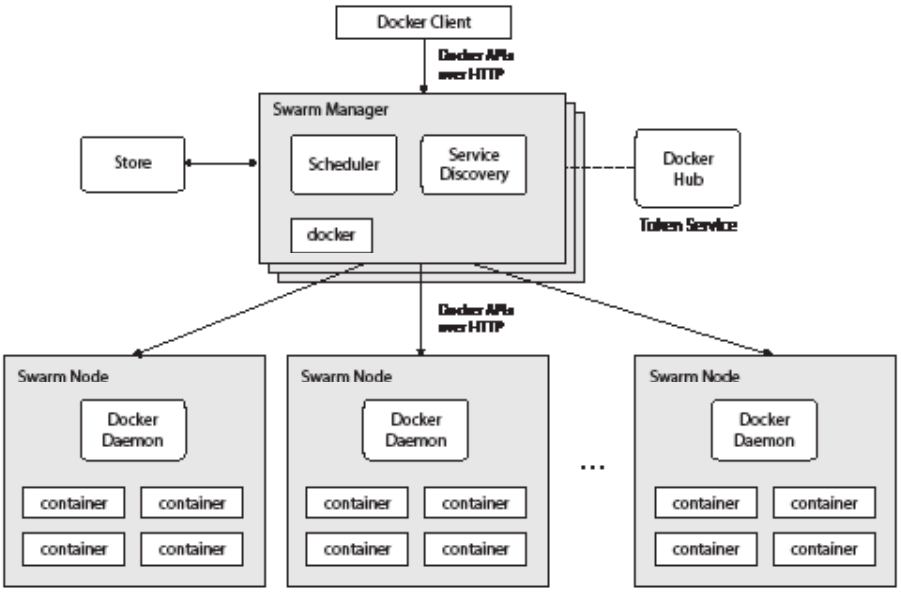

Figure 1: Docker Swarm Architecture

USABILITY

Docker Native Orchestration borrows concepts from single-node Docker and extends them to the Swarm. For users already familiar with Docker, this makes Swarm easy to learn and use. The same is true for application services. Docker users will already be familiar with Docker Compose. The docker-compose command accepts a YAML format file as input to describe a container or application service to be run on the Swarm. New extensions to Docker such as *Distributed Application Bundles (.dab files)* build on the familiar Docker Compose file format to support multi-service stacks.

With Docker Native Orchestration, Swarm has become a feature of Docker Engine

ORCHESTRATING APPLICATION SERVICES

The YAML syntax used in Docker Compose is continually evolving as features are added. A version number at the top of a Docker Compose file identifies the versions of Docker that a file is supported on depending on features used. At the time of this writing the latest version is 3.3. Older Compose files will run on the latest Docker Engine, but to take advantage of newer features, you'll require a recent version of Docker Engine.

The example below shows a *docker-compose.yml* file reflecting some new capabilities in Docker, specifically how services are deployed with configurable update policies and features like constraints. This example creates a service called *redis*, and instructs Docker to keep six instances of redis running across the Swarm. It also provides instructions that when updating the redis service, containers should be updated two at a time, ten seconds apart. The service also has the constraint that containers that comprise the service should be deployed only to worker nodes running Ubuntu 16.04.

```
version: '3'
services:
  redis:
    image: redis:alpine
    deploy:
      replicas: 6
      update_config:
        parallelism: 2
        delay: 10s
      restart_policy:
        condition: on-failure
      placement:
        constraints:
          - node.role == worker
          - engine.labels.operatingsystem == ubuntu 16.04
```

Before we provide more examples, now is a good time to digress and explain the notion of stacks, a new concept in Docker Swarm. While still experimental (as of Docker 17.03), Docker describes distributed application bundles in *DAB* format files. This allows applications comprised of multiple services to be deployed from a single file as a unit on a Swarm. These multi-service units are referred to as *stacks*.

Users can create a *DAB* file from existing Docker Compose files using *docker-compose bundle*. They can then use *docker deploy* to deploy the application stack.

Figure 2 illustrates the difference between an application service on a Swarm cluster (pictured at left) and a stack, comprised of multiple Docker services (pictured on the right).

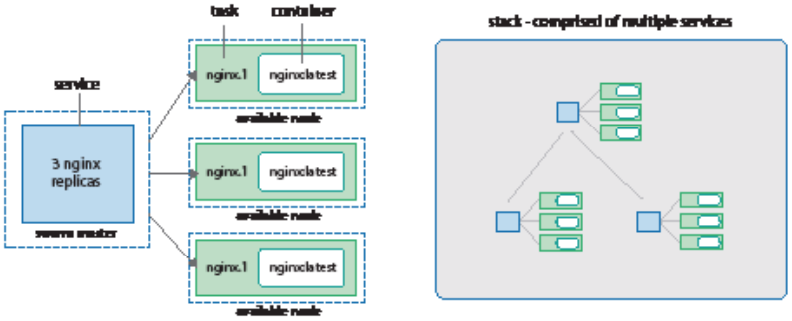

Figure 2: How services are defined in Swarm and combined into stacks

Stacks can be deployed directly from an existing compose file or the new DAB file format. This is a nice feature because it allows users to leverage existing compose files while still taking advantage of stacks.

```
$ docker deploy -c ./docker-compose.yml stack
Creating network stack_default
Creating service stack_redis
```

After deploying the stack, we can list the running services using *docker service ls*.

```
$ docker service ps stack_redis
ID            NAME           IMAGE         NODE                          DESIRED
STATE
2pogjmfh3a5c  stack_redis.1  redis:alpine  ip-172-31-45-41.ec2.internal   Running
ojlzs8mxf1go  stack_redis.2  redis:alpine  ip-172-31-31-222.ec2.internal  Running
so98zzxx0w0x  stack_redis.3  redis:alpine  ip-172-31-31-222.ec2.internal  Running
pkxoxyrnas97  stack_redis.4  redis:alpine  ip-172-31-14-45.ec2.internal   Running
uy4idzj1q180  stack_redis.5  redis:alpine  ip-172-31-26-208.ec2.internal  Running
btf6zecdmwof  stack_redis.6  redis:alpine  ip-172-31-2-103.ec2.internal   Running
```

We can scale our service on the swarm cluster by specifying a new target scale as shown:

```
$ docker service scale stack_redis=10
stack_redis scaled to 10
```

Swarm now supports multi-application, multi-host Distributed Application Bundles

After scaling, we see that that stack now has 10 replicas of the redis service:

```
$ docker stack services stack
ID            NAME         MODE        REPLICAS  IMAGE
hwbn3ppjt9s5  stack_redis  replicated  10/10     redis:alpine
```

SWARM FEATURES

Functionality in Swarm continues to improve with each release. Features in key areas of Swarm are described below. Later on, we'll look at Kubernetes in these same areas.

Networking

When services are deployed on a Swarm cluster, they are bound to a specified overlay network. If a network is not specified, they are bound to a default network. Swarm operates what's referred to as a "routing mesh". If a service is published to be available on port 8080 for example, every node in the cluster would listen on port 8080, and a container-aware routing mesh will route traffic to a node running one of the containers in the service. Load balancing is automatic, and is handled by Docker engine.

In Swarm, networking between nodes is auto-configured to use TLS encryption and mutual authorization. Management nodes host a certificate authority and certs are automatically rotated so that users don't need to worry about configuring them. An external certificate authority can be used as well.

It's worth mentioning that when Swarm is deployed in a Rancher environment, it is deployed with Rancher's own IPsec network that delivers similar functionality. When using Rancher's default "managed" network mode, containers are assigned both a Docker bridge IP address as well as a Rancher managed IP on the default docker0 bridge. Containers within the same environment are thus routable and reachable through the managed network.

Storage

Docker has a simple storage model that supports the use of data volumes. Data volumes are specified in a Docker Compose file or as an argument to the docker run command. Data volumes can be shared and reused among containers, and data volumes persist even if the container itself is deleted. Data volumes can also be mounted from a host directory on the underlying cluster node. A common practice is to use deploy a dedicated container that holds persistent sharable data resources, and allow other containers to reference the data from that storage container.

In Swarm environments, storage needs to be host independent to support multi-node deployments. In this case, Docker volume plug-ins can be used that allow you to provision and mount shared storage that is independent of a single Docker host (NFS, iSCSI or FC). An example of creating a volume backed by an NFS file system, and then mounting that volume into a container is provided below:

```
$ docker volume create \
    --name mynfs \
    --opt type=nfs \
    --opt device=:<nfs export path> \
    --opt o=addr=<nfs host> \
    mynfs
$ docker run -it -v mynfs:/foo centos sh
```

A variety of volume plugins are available for Swarm clusters including storage services such as AWS EBS, Cinder block storage (OpenStack) and CephFS.

Scheduling

In Swarm, the basic unit of scheduling is a task and each task maps to one container. You can provide guidance on where containers run by offering placement directions in a Docker Compose file or via the command line. Users can specify scheduling constraints based on things like a node id, hostname, role in the cluster or various user-defined node labels or Docker Engine labels. The scheduler will also consider resource requirements like cpu and memory, and balance containers appropriately to avoid nodes becoming over-subscribed.

The scheduler constantly monitors the desired state of application services running on the Swarm vs the actual state, and makes scheduling decisions to maintain the desired state – restarting a container, or starting a container on another node for example in the case of a host failure.

Service Discovery

Prior to Docker 1.12, when deploying Swarm clusters, it was necessary to register names along with node and node port combinations with an external service like consul, zookeeper or etcd. This is no longer required with Docker Engine operating in Swarm mode, because service discovery is now implemented natively.

In Swarm mode, service discovery is handled by an internal DNS component that assigns each service a virtual IP address and DNS entry in the Swarm overlay network. Containers share DNS mappings via an internal Gossip network and any container can access any other service by referencing its service name.

Load Balancing

As of Docker 1.12, load balancing is now also integral to Docker services. When services are deployed, each service gets an internal DNS name and a virtual IP address. Users don't need to worry about constructing their own load balancers, because container load balancing is implemented automatically using IP Virtual Server (IPVS). Users can still choose to use an external load balancer (ELB, HA-Proxy or Nginx) to distribute external traffic across Docker hosts if they prefer.

Scalability

Various sources, including a 2015 Docker blog article[1] indicate that Swarm clusters can scale to 1,000 nodes and 30,000 containers. Among these sources is a test conducted in the AWS cloud using a single Swarm manager (m4.xlarge) and 1,000 (t2.micro) nodes with 30 containers per node[2] . This test was conducted in Swarm standalone mode, but subsequent benchmarks have confirm that Swarm scales to large numbers of nodes and containers.

At present, Swarm does not support the notion of auto-scaling. While advanced users may be able to script this functionality themselves, there is no direct support in Swarm.

It's worth noting that just because a cluster can physically scale to a large number of nodes, doesn't mean it can be efficiently managed at that scale. Clusters may be augmented with additional monitoring or management tools to assist with management at scale.

Performance

While the performance of applications is often unrelated to the performance of the orchestration framework, for some applications that need to wait for containers to start, or API calls to complete, performance can be an important consideration. Swarm appears to exhibit excellent performance for a variety of orchestration and scheduling functions. An independent benchmark[3] done by Jack Nickoloff using a recent version of Docker Swarm (1.1.3) measured relative performance between Swarm and an earlier version of Kubernetes (1.2.0-alpha7). In objective comparisons related to the time required to start containers and synchronous commands like listing containers, Swarm outperformed Kubernetes by a wide margin, especially as the clusters became busy. Performance results results showed that an impressive 99% of all API calls responding within 360 milliseconds on a 1,000 node cluster.

In March of 2017, the Kubernetes blog detailed a much larger 5,000 node benchmark[4] conducted with Kubernetes 1.6 suggesting large strides have been made in Kubernetes scheduling performance as well. Unfortunately, an objective benchmark has not been published since 2016, but both Kubernetes and Docker Swarm appear to be making significant strides in performance with Kubernetes closing the gap in what appears to be a slight advantage for Swarm.

For users already familiar with Docker, Swarm is intuitive, leveraging the familiar Docker command set and docker-compose file format

Overall Impressions

It's hard not to be impressed with the elegance of Docker Swarm. It's fast, responsive, and has an elegant design and command set. Swarm now supports multi-service, multi-host application deployments like Kubernetes, functionality that didn't exist just a year ago.

Docker is increasingly becoming an "all-in-one" framework. Docker Community Edition remains open source, but with its new Enterprise Edition, Cloud Service and Docker Store (augmenting Docker Hub), Docker is understandably making moves to draw enterprise users into Docker's own ecosystem of products and services.

For users who are comfortable with Docker Compose and the Docker command set, and who prefer an easy-to-use solution, Swarm is a logical choice. The feature set is smaller than Kubernetes however, and Swarm lacks some of Kubernetes' advanced features.

KUBERNETES

Kubernetes is a second popular orchestration framework for container workloads. Kubernetes is based on Google's well-regarded Borg system that has been managing containerized workloads at scale for over a decade. This heritage gives Kubernetes credibility with administrators managing large clusters and workloads.

Kubernetes was spun off as open-source project in 2015 by Google, and the technology has been rapidly evolving. While it's young as open-source projects go, the underlying architecture is mature and proven. The name Kubernetes derives from the Greek word for "helmsman", and is meant to be evocative of steering container-laden ships through choppy seas.

Conceptually, Kubernetes is like Swarm in that it deploys a cluster with master nodes and worker nodes and employs a RAFT algorithm for consensus among masters, but this is where the similarity ends. Unlike Swarm which is now basically a feature of Docker, Kubernetes is its own distinct environment.

Some notable differences between Swarm and Kubernetes are as follows:

- Kubernetes has the reputation of being difficult to install and manage. It relies on external services like etcd to share state among Kubernetes nodes, and requires that a separate overlay network be installed and configured. Unlike Swarm, Kubernetes clusters generally require careful, advanced planning.
- While there are multiple ways to deploy a Swarm cluster, there are an *enormous* number of deployment options for Kubernetes[5]. Choices range from local-machine installations to turnkey cloud solutions, to various custom solutions targeting on-cloud, on-premises, VM-based or bare metal deployments. Each solution may support different operating environments, cloud providers, configuration management and networking solutions and load balancing options. Solutions vary depending on the provider. This can make it complex to evaluate Kubernetes because the software components used can vary depending on the Kubernetes distribution.

While powerful, Kubernetes requires administrators to learn a number of new concepts and master a new command line interface

- Kubernetes has its own command line interface (*kubectl*) and its own API distinct from Docker representing a learning curve for users already familiar with Docker.
- Kubernetes does not use Docker Compose format, rather it has its own file specification (also YAML based) for deploying containerized application services.
- Kubernetes employs a different management model than Swarm. Rather than the basic unit of scheduling being a task or container, Kubernetes introduces the notion of *Pods*.

A Primer on Kubernetes

Kubernetes involves some concepts that at first glance may seem confusing, but for multi-tier applications, the Kubernetes management model is proven and powerful.

The basic workload unit in Kubernetes is a *Pod*, a collection of one or more containers that reside on the same cluster host. Pods are managed by *Replication Controllers* associated with each application *Deployment*. Replication Controllers facilitate horizontal scaling and ensure that Pods are resilient in case of host or application failures. The Replication Controller is responsible for ensuring the desired number of Pods are running on the cluster and liaising with the scheduler to place them optimally. If a container goes down or a host becomes unavailable, Pods will re-start on different hosts as necessary to maintain a target number of replicas.

In Kubernetes, the notion of a Service also has a slightly different meaning than in Docker Swarm. Services in Kubernetes are essentially load balancers and front-ends to a collection of Pods. A Services' IP address remains stable and can be exposed to the outside world via an Ingress, abstracting away the number of Pods as well as virtual IP addresses for each Pod that can change as Pods are scheduled to different cluster hosts. This all sounds a little complicated, but basically a Deployment described in a Kubernetes YAML file can define the Pods, Replica Sets and the Services that comprise an application all at once (as well as other things like Networks, Volumes, Secrets, Ingresses etc.).

Kubernetes uses a flat networking model. It doesn't configure a network routing mesh like Docker Swarm (making services reachable from outside the cluster), but similar functionality can be configured optionally using Kubernetes *NodePort* functionality to expose services on high-numbered ports on each cluster node and route to internal services. More commonly, external applications will connect to an Ingress IP address which in turn will direct traffic to the service and the underlying Pods that comprise the application. Kubernetes provides multiple choices as to how services are exposed externally.

Kubernetes does not specifically use the term "stack" to describe multi-service, multi-host deployments but the notion of a stack is implicit in a Kubernetes deployment.

ARCHITECTURE

The architecture of Kubernetes is more complicated than Docker Swarm. This is because Kubernetes has more components and relies on external services as explained previously like etcd (a distributed key-value store) and a user-provided overlay network like *flannel* or *weave*.

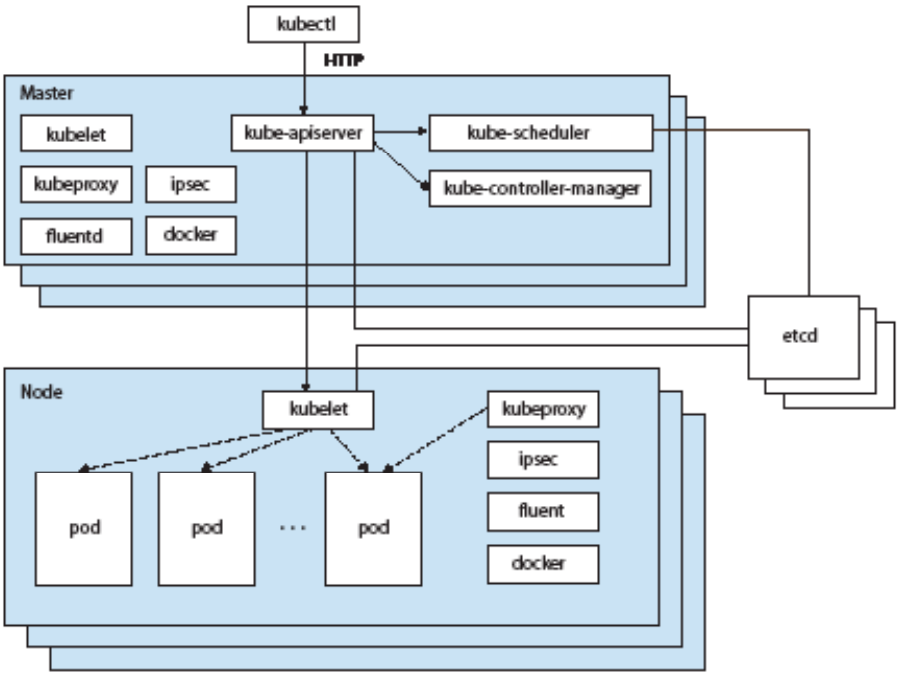

Figure 3: Kubernetes Architecture

KUBERNETES COMPONENTS

Figure 3 shows the architecture of a Kubernetes cluster. The major components are described below.

- **kubeapi-server** exposes the Kubernetes API. The API server is designed to scale horizontally by deploying more instances as the cluster grows.
- **etcd** is the backing store where cluster metadata is stored in key-value format.
- **kube-controller-manager** is a single binary that runs different controller functions including the node controller, replication controller, endpoints controller and service account & token controllers.
- **kube-scheduler** watches for newly created Pods that have yet to be assigned, and selects a node for them to run on.
- **kubelet** is the primary node agent. It manages Pods on the local node running the Pod's containers, mountingPpod volumes, reporting Pod status, monitoring node status etc.
- **kube-proxy** enables the Kubernetes service abstraction by maintaining network rules on the host and performing connection forwarding.

Support for application deployments is mature in Kubernetes, with many sophisticated features

- **docker** is used for running containers on each host. Needless to say, when running docker with Kubernetes, swarm-mode is not enabled.
- **fluentd** is a daemon that helps implement cluster-level logging.
- **ipsec** Rancher IPsec network-based overlay. This is the network overlay used in Rancher. Other Kubernetes implementations may use an alternative network overlay.

USABILITY

Kubernetes exposes its own CLI as well as a comprehensive web interface. It also uses its own YAML or JSON format syntax to specify multi-service applications that can be deployed on the Kubernetes cluster. The different formats used by Kubernetes will be an issue for users that prefer to user docker-compose format files, although there are tools available that convert from docker-compose to Kubernetes YAML format.

ORCHESTRATING APPLICATION SERVICES

To illustrate orchestration, we use one of Google's sample Kubernetes applications called guestbook available on the Kubernetes GitHub site. This is a good example because it is a scaled-out, multi-tier application that uses redis as a distributed datastore. A simple YAML specification file (guestbook-all-in-one.yaml) defines the application and its various tiers (a front-end web tier, a redis master and multiple redis slaves).

Similar to using "*docker stack deploy*" in Swarm, the full application stack can be deployed using a single command with Kubernetes:

```
$ kubectl create -f /path/guestbook/all-in-one/guestbook-all-in-one.yaml
service “redis-master” created
deployment “redis-master” created
service “redis-slave” created
deployment “redis-slave” created
service “frontend” created
```

The application is comprised of multiple deployments as shown.

```
> kubectl get deployments
NAME           DESIRED   CURRENT   UP-TO-DATE   AVAILABLE   AGE
frontend       3         3         3            0           5m
redis-master   1         1         1            0           5m
redis-slave    2         2         2            0           5m
```

Users can also choose to use the Kubernetes Dashboard to deploy an application. If Rancher is used to deploy Kubernetes, the Dashboard is installed automatically saving administrators the trouble of installing and configuring it separately.

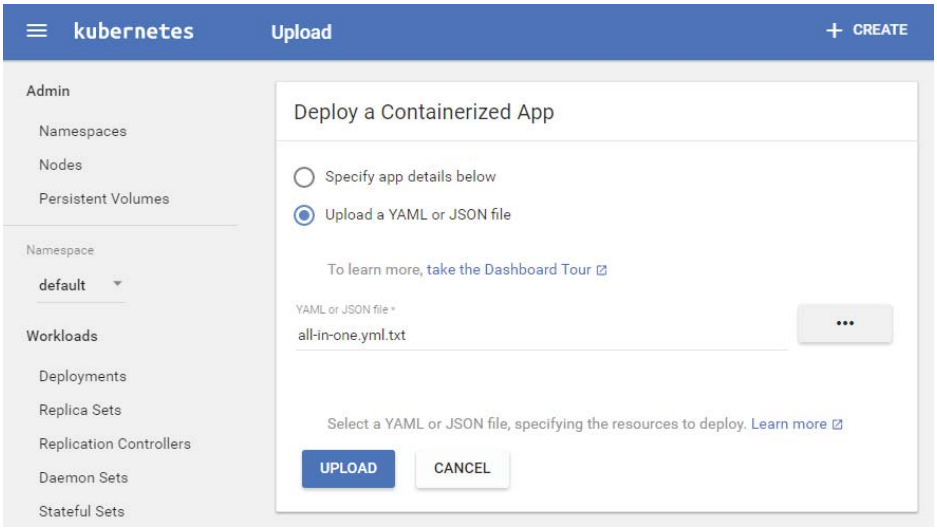

Figure 4: Deploying a containerized app using the Kubernetes Dashboard

Support for distributed applications is rich in Kubernetes. Unlike Docker that uses a few simple concepts (Containers, Services and Stacks) Kubernetes is more granular, exposing multiple types of resources:

- **Pods** – Basically the unit of scheduling in Kubernetes, a collection of one of more containers that provides some basic functionality for an application
- **ReplicaSets** – A controller construct in Kubernetes that ensures the correct number of Pods remain running for each application
- **Services** – An abstraction in Kubernetes that groups a local set of Pods providing load balancing, a single endpoint, and abstracts details like the number of Pods in the services and their individual virtual IPs
- **Ingress** – A separate resource that maps externally accessible IP addresses and ports to a Kubernetes service

Figure 5 illustrates these Kubernetes concepts with our deployed guestbook application. The application is comprised of multiple Replica Sets, each responsible for managing the Pods that comprise various tiers of the application. The result is similar to a Docker Swarm Stack, but the terminology and architecture is different.

Kubernetes has an easy to use web dashboard directly accessible from within Rancher

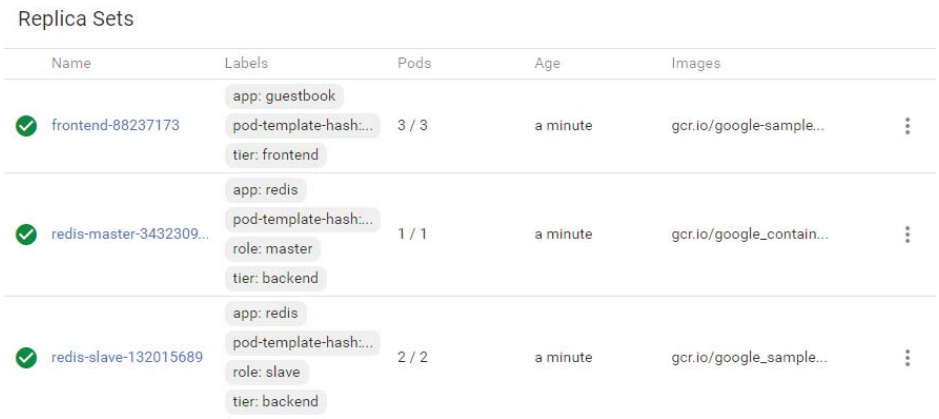

Figure 5: A view of the deployed guestbook application in the Kubernetes

FEATURE SET

The feature set in Kubernetes is large. While It's not possible to explore all the features here, some of the major components are described below.

Networking

To understand networking in Kubernetes, it's useful to contrast it with the approach taken in Docker and Docker Swarm. By default, Docker uses host private networking. A virtual bridge is created on each Docker machine (docker0) and when containers are created, a virtual ethernet device in each container is connected to the host bridge. The result is that each container has a unique IP address on the docker0 bridge, and that containers can talk to one another on a single host. Therefore, in both Docker Swarm and in Kubernetes, it is necessary to create an overlay network to allow containers to communicate with one another across hosts. In Docker Swarm, the overlay network is part of the Docker Swarm functionality.

Kubernetes does not natively implement a network solution for communication across cluster hosts, but it does impose some requirements on the networking solution used. Among these are:

- All containers must be able to talk to all other containers without NAT
- All nodes need to be able to talk to all containers (and vice versa) without NAT
- Containers see themselves as having the same IP address as other entities on the cluster

This model, when implemented, is intuitive for both developers and cluster administrators. It means that complexities around network routing don't get in the way of application deployments. A key difference in the Kubernetes model is that IP addresses in Kubernetes are applied at the Pod level rather than at the individual container level. This means that containers within a Pod communicate with one another via localhost and must coordinate port usage between containers in a Pod share a common IP address.

Kubernetes provides administrators with flexibility to deploy different distributed key-value stores and software-defined network solutions

Kubernetes supports the Container Network Interface (CNI) standard[6], a network plug-in architecture that allows you to use a variety of software-defined network (SDN) environments. There are no less than a dozen possible network solutions described in the Kubernetes documentation including:

- **Cilium** – an open-source project that secures networking between containers
- **Contiv** – an open-source project configurable to support various networking use cases
- **Contrail/OpenContrail** – a multi-cloud network virtualization solution often integration with platforms like Kubernetes, OpenStack and Mesos
- **Flannel** – a simple overlay network that satisfies the Kubernetes requirements above
- **Weave Net** from Weaveworks, and easy to use network for Kubernetes
- **Other CNI network plug-ins including Rancher's**

Installing Kubernetes in a Rancher environment simplifies networking because Rancher provides its own IPsec network overlay on the cluster. This native network enabled by default in the environment templates for both Swarm and Kubernetes. The network service manifests itself in Kubernetes as two infrastructure stacks visible to administrators, one called *ipsec* and the other called *network-services* that run across the cluster. By default, all containers that are provisioned join the managed network.

IP addresses in Kubernetes for Pods and Services are routable only within a Kubernetes cluster and are not exposed externally. This means that to expose a Kubernetes service on an externally accessible IP address and port, an Ingress or other method of exposing an external address is required. Kubernetes administrators can setup as many Ingresses as they would like on a cluster. Like other resources, the definition for ingresses are specified in the YAML definition of the application. A simple example is below.

```
apiVersion: extensions/v1beta1
kind: Ingress
metadata:
  name: my-ingress
spec:
  backend:
    serviceName: testsvc
    servicePort: 80
```

If you create the Ingress above using *kubectl create -f* you should see:

```
$ kubectl get ing
NAME                RULE          BACKEND        ADDRESS
my-ingress          -             testsvc:80     107.178.254.228
```

The Ingress structure above is referred to as a simple ingress where an external IP addressed is mapped directly to a Kubernetes Service. Load balancing among Pods is implement by the Service. There are other types of Ingresses as well including a Fanout Ingress that connect to multiple back-end Services.

Storage

In terms of their implementation, Kubernetes volumes are basically directories accessible to containers in a Pod. Each container in a Pod must explicitly mount a shared volume for the volume to be accessible to the container, and different containers can have different mount points. How the data in the volume comes to exist, and where the data physically resides, depends on the type of Kubernetes volume.

There are over a dozen types of volumes supported in Kubernetes, providing cluster administrators with flexibility depending on their environment. Volumes types that can be specified in Kubernetes include *emptyDir*, *hostPath*, *nfs* and *persistentVolumeClaim*. If your cluster is deployed in a public cloud environment, there are also volume types for popular public cloud storage including *azureFileVolume*, *gcePersistentDisk* and *awsBlockStore*. There are also volume types specific to distributed data stores such as *flocker*, *glusterfs*, *cephs* and others.

A simple example showing how a data volume is associated with a Pod is provided below. In this example, the Pod includes a definition for a volume called *data-volume* of the type *emptyDir*, meaning that the contents of the volume are initially empty, but that any data stored there will persist for the life of the Pod. In one of the containers associated with the Pod (my-container) the volume is mounted under */data*. Any of the types of volumes are specified in a similar manner in Kubernetes including various types of persistent data volumes.

```
apiVersion: v1
kind: Pod
metadata:
  name: my-pod
spec:
  containers:
  - image: gcr.io/google_containers/my-webserver
    name: my-container
    volumeMounts:
    - mountPath: /data
      name: data-volume
  volumes:
  - name: data-volume
    emptyDir: {}
```

Scheduling is a strength of Kubernetes, with multiple policies and support for multiple workload types

Scheduling

Kubernetes supports a broad set of workload types, and rich scheduling functionality. It supports the use of user-defined custom schedulers, a beta feature introduced in Kubernetes 1.6.

Whereas Swarm schedules containers as tasks, Kubernetes schedules Pods, collections of one or more containers residing on a single Docker host. A variety of controllers exist in Kubernetes that work with the scheduler to support various workload patterns. Some of these patterns like Replica Sets and Deployments we've already covered. Other types of deployment patterns in Kubernetes' repertoire include:

StatefulSets – Previously referred to as "PetSets", StatefulSets are useful for Services that require special handling. This may be because they require stable network identifiers, stable storage, or because there are requirements around the order in which services are deployed, shutdown or scaled. StatefulSets make it possible to deploy a broader set of workloads by accommodating specialized requirements. This concept does not exist in Swarm, where workloads with these requirements would need to be placed outside the Swarm cluster.

DaemonSets – Daemon sets are similar to the notion of a Global Service in Swarm, except instead of a container being orchestrated to each cluster node, Kubernetes will run a Pod on each cluster host. The idea is similar however.

Jobs (that run to completion) – The notion of a job that runs to completion is similar to jobs in traditional, non-container workload schedulers like IBM Platform LSF or Altair's PBS Professional. Basically, a Job creates a collection of Pods and coordinates with the scheduler to makes sure that each Pod in the Job successfully completes. Jobs can be parallel, non-parallel or parallel associated with a work-queue. While not implemented yet, according to the Kubernetes documentation, there is also the notion (planned) for being able to pass each parallel job an index, thus providing "array job" functionality useful for various types of parametric simulations. These features make Kubernetes suitable for running various types of batch workloads. This is useful for users that wish to mix persistent service-oriented workloads and batch workloads (implemented in containers) on the same shared cluster.

Cron Jobs – Cron Jobs are a special type of job that can be scheduled to run at a specific date and time or based on a particular recurrence pattern. Like other Kubernetes workloads, Cron Jobs can be defined in YAML format specifications or launched from the command line as shown:

```
$ kubectl run hello --schedule="*/1 * * * *" --restart=OnFailure --image=busybox
-- /bin/sh -c "date; echo This message appears every minute"
cronjob "hello" created
```

The schedule above takes a cron format string representing the minute, hour, day of month and day of week expressed in numerical formats so that complex recurrence schedules can be expressed. The syntax "*/1" in the first field corresponds to "once every minute".

Load Balancing

There are multiple ways of achieving load balancing with Kubernetes. The most obvious way is to deploy a service comprised of Pods sharing the same label. For each service, an internal cluster IP and port is available, and Kubernetes will automatically load balance requests coming to that port. This IP address is accessible only inside the cluster however.

When presenting a service externally, there are multiple approaches. One solution is *NodePort*, a solution that exposes a high-level port externally on every cluster node routable to the service. This is not ideal though, because it is more common to expose external services on lower numbered ports, and a NodePort does not allow this flexibility. Another solution is to use an external LoadBalancer such as those available with GCE or AWS. The Load Balancer can bypass the service and point directly to the virtual IP addresses of the Pods that comprise the service. Load Balancers vary by cloud provider.

The most common solution when exposing services externally is to use an Ingress (a collection of rules that describe how to reach cluster services) coupled with an IngressController. Examples of Ingress Controllers are HAproxy, Vulcan and Nginx. Typically, when a service is deployed that needs to be accessible externally, an Ingress will be defined that specifies the publicly accessibly port and IP address that maps to the back-end service.

Service Discovery

A service registry is responsible for maintaining a database of services and providing an API (HTTP or DNS) that clients can interact with. In Kubernetes, the key-value store used for the service registry is implemented using *etcd*, a distributed key-value store. Service discovery in Kubernetes is comprised of the following components:

- A DNS service used to map service names to IP addresses (*kube-dns*)
- etcd, the key-value store for the service database
- A kubelet process running on each host responsible for health checks. The Replication Controller takes care of maintaining the target pod count
- kube-proxy, responsible for load balancing traffic to individual pods

Kubernetes provides multiple options for service discovery depending on whether a client is internal or external to the cluster. For internal Service discovery, Kubernetes' internal DNS service is used. Every service registers to the DNS service allowing services to find and talk to one another.

For external service discovery, there are multiple options.

- **NodePort**: In this method discussed earlier, Kubernetes exposes the service through special ports (30000 to 32767) of a node's IP address
- **LoadBalancer:** This approach is implemented by cloud providers and allows a LoadBalancer to redirect traffics directly to the pods that comprise a service
- **Ingress:** Described above, an Ingress can present an external IP address and port number that links to a Kubernetes service.

Both Swarm and Kubernetes have made significant strides in scalability and performance

Scalability

When it comes to scalability, Kubernetes' Google heritage provides credibility. Various sources describe multiple Kubernetes environments in the range of 15,000 nodes, however these environments typically run multiple logical clusters. As of Kubernetes 1.3, the service-level objective (SLO) on a fully loaded 2,000 node cluster was to provide a guaranteed response time of less than one second for any API call.

In a March 2017 blog article[7], additional guidance was offered about scalability in Kubernetes along with a compelling benchmark demonstrating scalability. Using Kubernetes 1.6, a 5,000 node cluster with 150,000 pods was deployed on AWS to demonstrate improvements in average start-up latency and API response times. Kubernetes architects have estimated that with recent advances in Kubernetes, clusters in the range of 20,000 hosts[8] could meet one second API response time SLO referenced above.

As we explained when describing scalability in Docker Swarm, real scalability is about more than how many nodes or containers can simply be deployed. Supporting an environment at scale often depends on having the flexibility to solve real problems and automate tedious tasks to minimize the need for manual interventions. With its broad feature set, Kubernetes provides administrators with more tools to manage a large-scale application environment.

Performance

In the large scale 5,000 node Kubernetes benchmark referenced above, Pod startup latency was measured to be slightly over one second on average with 99% of Pods starting within 1.9 seconds. This result obtained with Kubernetes 1.6 is a significant improvement over an earlier benchmark done by Jack Nickoloff[9] comparing the relative performance of Swarm and Kubernetes scheduling. Unfortunately, there is no recent benchmark that compare the two frameworks objectively, but Kubernetes appears to be making significant strides in performance.

Autoscaling

While Swarm allows manual scaling of services, Kubernetes supports dynamic autoscaling in response to dynamic changes in application demand. Kubernetes will also consider the state of various cluster nodes and other workloads to optimize placement. An example of autoscaling a Kubernetes deployment is shown below. In this case we ask the scheduler to manage our deployment called *frontend* to target 80% cpu utilization with the constraint that a maximum of 10 nodes can be deployed.

```
$ kubectl autoscale deployment frontend --max=10 --cpu-percent=80
```

Kubernetes offers additional features not available in Docker Swarm including:

- **Container Lifecycle Hooks** (for building cluster-aware applications) – for example a "PostStart" hook can be configured to execute a command in the context of a container immediately after it is started. A similar "PreStop" hook allows custom code to be injected to possibly clean things up before a container is terminated.
- **Guaranteed Scheduling** (for critical pods with SLA guarantees) – for application services that are critical, Kubernetes supports a "Rescheduler" that can attempt to free up space by evicting less critical Pods from the cluster, essentially providing workload preemption and ensure that critical services run.

OVERALL IMPRESSIONS

While Kubernetes lacks the seamless integration with Docker offered by Swarm, the functionality is richer in several areas. Comparing one feature like Health Checks in both orchestration engines in more detail helps illustrate this point.

In Swarm, Health Checks are specified in a docker-compose or DAB file. A scriptable instruction (a curl command for example) is used to implement the Health Check and decide whether a container is healthy or unhealthy. Docker exposes three options aside from the command itself - the interval between checks, the timeout after which a Health Check is judged to have failed, and the number of times a Health Check will retry before reaching a conclusion.

In Kubernetes, the approach is similar, but as with many things in Kubernetes, more involved. "Liveness Probes" as they're called in Kubernetes can be specified similar to Swarm, but there are different types of probes:

- An **exec probe** involves a user-defined command that executes inside the container. Similar to Swarm, the command is executed at a configurable period, but Kubernetes additionally allows the specification of an initial delay period as well.
- A separate type of Liveness Probe is configurable called a **httpGet probe** that will check a URL path and port and send httpHeaders with configurable values. The health of the container will be judged based on the returned HTTP code.
- A third type of **tcpSocket probe** attempts to open a TCP socket connection on a particular port to judge whether a containerized service is live and responding.

In addition to Liveness Probes, Kubernetes also has the notion of "Readiness Probes". This reflects the real-world scenario where sometimes containers are "alive", but they are too busy doing other work to respond to "Liveness Probes". For example, a container may be loading a large dataset or doing cleanup, and should not be killed just because it is unable to respond to a Liveness Probe. In addition to differentiating between readiness and liveness, and supporting multiple types of liveness probes, Kubernetes allows more configurability supporting things like initial delays, success thresholds (minimum consecutive successes for a service to be considered healthy) and failure thresholds (number of consecutive failures before a service is considered unhealthy). This is just one example where both orchestration solutions "tick the box" and have the basic functionality, but the implementation in Kubernetes is richer and more configurable.

COMPARING ORCHESTRATION ENGINES

Both Swarm and Kubernetes are capable orchestration engines for deploying and managing containerized applications. While they provide similar feature sets, they differ in architecture and implementation.

To help understand the differences, table I below explores some functional areas of both orchestration solutions and describes how they are implemented in each framework.

Area of comparison	Docker native orchestration (swarm)	Kubernetes
Relationship to Docker	Swarm is now part of the Docker distribution. The Docker command set has been extended to support multi-application, multi-host deployments	Kubernetes is a distinct orchestration framework from Docker. It has a separate command line facility and Web dashboard. While it relies on Docker services, it supports application types and scheduling policies not readily supported in Docker
Installation experience	Swarm clusters are relatively easy to install, scale and manage. Cluster nodes can be easily added and the environment is largely self-contained and self-configuring	Kubernetes has the reputation of being difficult to install and manage. It depends on external services that need to be installed and managed separately. Rancher mitigates these challenges however automating installation and management.
Ease of use	For users already familiar with Docker, Docker's native orchestration will feel intuitive and easy to use.	Using Kubernetes requires that user master new concepts and learn a new CLI that while quite rich is different than the Docker CLI. Kubernetes does provide advanced features however that for seasoned administrators can actually make the environment easier to manage. (ensuring SLAs, autoscaling services etc.)
Application templates	Swarm uses existing docker compose templates (based on YAML) so templates move seamlessly into a Swarm environment. Docker also has a separate DAB file format (Distributed Application Bundles) similar in syntax to Compose but meant to define application "stacks"	Uses YAML (or JSON) templates with a format specific to Kubernetes. While there are some tools that do translation from Docker Compose, Kubernetes users will need to learn the new format. Kubernetes' application specification natively supports the notion of stacks so users have only one format to deal with.

Area of comparison	Docker native orchestration (swarm)	Kubernetes
Concepts	There are a relatively small number of concepts to learn when managing a Swarm cluster. Users need to understand things like containers, tasks, services and stacks.	Kubernetes has a richer set of concepts than Docker including constructs like pods, deployments, replica sets, daemon sets, ingresses, jobs and a variety of other features. While more complex, the more granular configurability in Kubernetes can make it easier for administrators to get the behaviors they want.
Unit of Scheduling	A Task – referring to a single Docker container and specific commands that run inside the container.	A Pod, an entity that runs on a single host but is comprised of one or multiple containers performing related functions.
Management interface	Uses the Docker CLI. While Docker does not have its own native Web UI, there multiple open-source solutions (Shipyard, Portainer etc.) Rancher exposes Portainer when Swarm is installed.	Separate Kubectl command distinct from Docker. Kubernetes also has its own dashboard which is intuitive and easy to use automatically exposed in Rancher
Best suited for	Simpler deployments, but the feature set in Swarm is growing rapidly allowing it to support larger applications and a wider variety of use cases	Larger production environments with dedicated cluster administrators. Kubernetes generally has stronger facilities to support multiple groups, namespaces and SLAs making it a better solution for multi-tenant clusters.
Rolling updates	Updates are achieved by telling the scheduler to use a new image. (docker service update –image) Updates can be rolled out in stages with configurable delay and parallelism. Updates can be rolled back.	Kubernetes handles updates one node at a time, using more configurable healthchecks (liveness checks) to verify proper functionality. Failed updates can be automatically rolled back
Data volumes	Docker data volumes are directories shared within one or more containers – volumes are native to the node they are created on. For global volumes, Docker relies on separate volume plugins.	Volumes are abstraction that allow containers to share data within the same pod. External data volume managers can be persistent and supported sharing data between pods. Kubernetes supports a wide variety of volume types providing flexibility to application architects and cluster administrators

Area of comparison	Docker native orchestration (swarm)	Kubernetes
Approach to networking	Swarm deploys its own network and implements what is known as a "routing mesh" allowing all nodes in the swarm to accept service connections, and route connections even when services are deployed on other cluster nodes. Users specify overlay networks when they deploy applications in Swarm.	Kubernetes does not include its own network framework, but imposes requirements on how the software-defined network needs to operate. There are over a dozen distinct network solutions used with Kubernetes including choices like flannel and weave. Rancher provides its own IPSEC based network solution that is feature rich
Security / TLS network authentication	Swarm is secure by default. Connections between nodes are automatically secured through TLS authentication with certificates. Rancher uses its own IPSEC solution with Swarm and offers secure communication between nodes, containers and services.	Ease of setup depends on the network framework chosen. TLS authentication is possible, but requires that certificates be generated and installed on each node. Solutions like Rancher greatly simplify Kubernetes network administration.
Secret management	Docker secrets are supported in swarm mode only. Secrets are stored in the encrypted RAFT log (as of Docker 1.13) – when a service is granted access to a secret, the secret appears in an in-memory file system at */run/secrets/secret-name* – Docker 1.13 includes functionality to lock the TLS encryption key (swarm lock)	Secrets are supported natively in Kubernetes – secrets can be passed to containers in volumes or via environment variables by adjusting the YAML used to create application services
Scaling	For each service, you declare the number of tasks to run. When you scale up or down, the swarm manager automatically adds or removes tasks to maintain a desired state	Kubernetes handles scaling similarly, but scales pods rather than simple tasks. Scaling is handled by a Replica Set policy. Kubernetes supports auto-scaling policies as well, a capability not available in Swarm

The choice or orchestration framework can impact on how developers build their applications

CONSIDERATIONS FOR APPLICATION ARCHITECTS

While the choice of framework has implications for operations teams, architects and developers are affected as well. Developers can design containerized applications to be largely agnostic of the orchestration solution used, but at some level, applications need to be aware of the environment. Also, developers may want to take advantage of orchestrator specific features.

- Developers building Pods for Kubernetes may want to design their applications accordingly. Deploying Pods comprised of multiple containers will have implications for how the containers communicate with one another since containers in Pods all have the same IP address and communicate via localhost. Applications built for this environment may not be readily portable to Docker Swarm where the notion of a Pod don't exist.
- How credentials are passed to application containers is something that developers will need to consider as well. For example, a login and password to a database instance may be passed as a secret in a mounted volume. In Kubernetes, secrets may be passed to a Pod as an environment variable instead. Developers will need to be aware of these differences.
- Developers may design their application services such that they are "cluster-aware", and reach outside the application domain and use the orchestrator's REST API to make applications self-monitoring and managing, introducing additional dependencies on the chosen framework.
- Appropriate compose files, DAB files, or Kubernetes YAML specifications will need to be written depending on where the applications will run.
- Application designers will need to think about how they test for the health of their application services, and decide whether they want to take advantage of Kubernetes specific constructs like Readiness Checks and Liveness Checks vs. Swarm's simpler notion of Health Checks. Developers might also use Rancher's own Health Checks.
- Applications may rely on services have particular needs where Kubernetes' Stateful Sets can be a useful feature. In Swarm deployments, it may be necessary to host these types of special services outside of the cluster.

For the reasons above, it's important that both application teams and operation teams have a voice in selecting the orchestration framework. For simple applications, it may not make a difference, but as applications become more complex, implementation approaches will be affected by the choice or orchestration engine and vice versa.

Rancher simplifies deployment and management for both Swarm and Kubernetes environments allowing users their choice of orchestration framework

ORCHESTRATION FRAMEWORKS AND RANCHER

Rancher makes it easy to deploy and run Docker containers in production. Rancher supports both Kubernetes and Docker Swarm along with other frameworks including Mesos, Windows containers and Cattle, Rancher's own native orchestration engine. It also provides a common set of infrastructure services that make it easy to deploy and manage clusters on premises and in various cloud environments.

Rancher supports multiple orchestration engines concurrently on different sets of infrastructure hosts grouped into distinct environments. Environments are added through the Rancher UI as shown in Figure 6 below. To add a new environment, users select an Environment Template, and begin adding hosts to the cluster. Adding a new Linux infrastructure host is as simple as copying a command provided from within the Rancher UI and pasting it into a shell on an infrastructure host to add the Rancher agent.

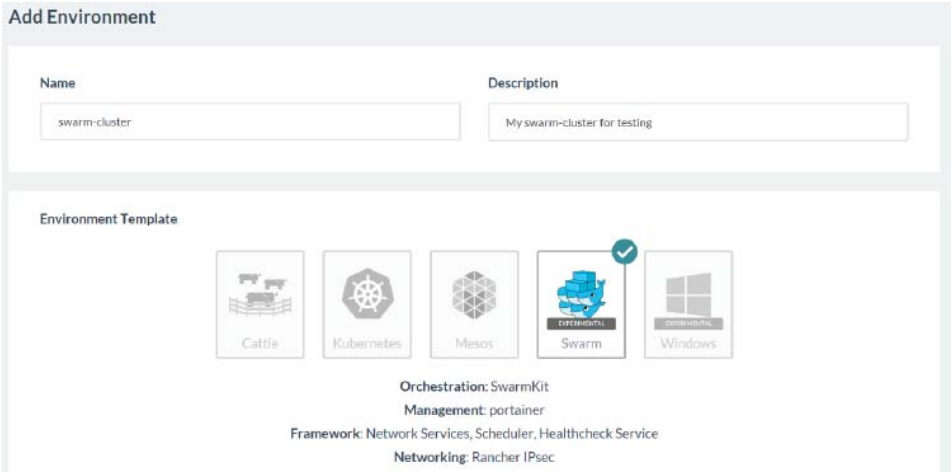

Figure 6: Deploying a Docker Swarm environment in Rancher

Adding cloud hosts in even more straightforward. Users can select from a menu of popular cloud providers (AWS, Azure, Rackspace and others), specify a few details like cloud specific credentials, regions and node types, and watch as Rancher automatically provisions a ready to run cluster. An Advanced Settings area allows fine grained control over a variety of installation options including the version of Docker installed on cloud hosts.

Rancher sets up a Swarm environment easily, and includes a web interface that can be used to manage the Swarm called Portainer (http://portainer.io). Once the cluster is installed and functional, Portainer is accessible under the Swarm pull down menu in Rancher.

Regardless of the orchestration solution used, The Rancher Web UI makes cluster and application management intuitive

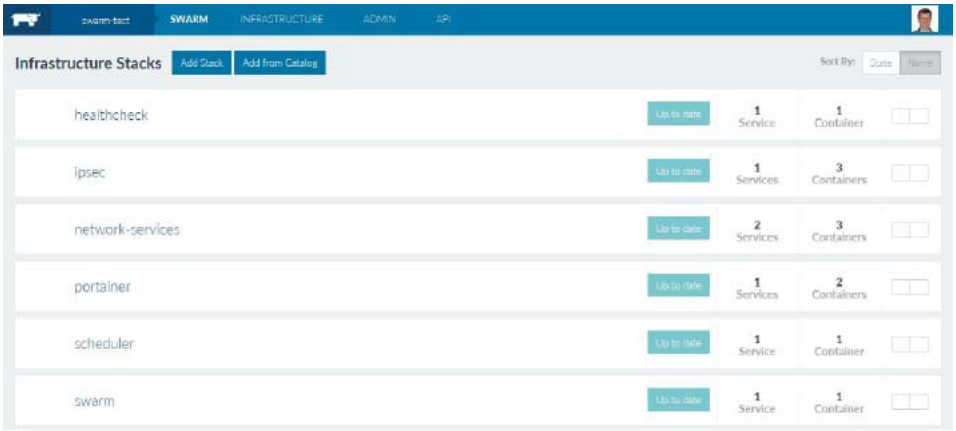

Figure 7: Swam is automatically installed in Rancher and components appear as infrastructure stacks

Kubernetes has a reputation for being challenging to install and configure, in part because it relies on external components. The Kubernetes environment template in Rancher makes it straightforward to install Kubernetes on both public cloud environments or servers running Docker on premises. Rancher exposes a shell with the *kubectl* command line as a pull down accessible in the Rancher UI.

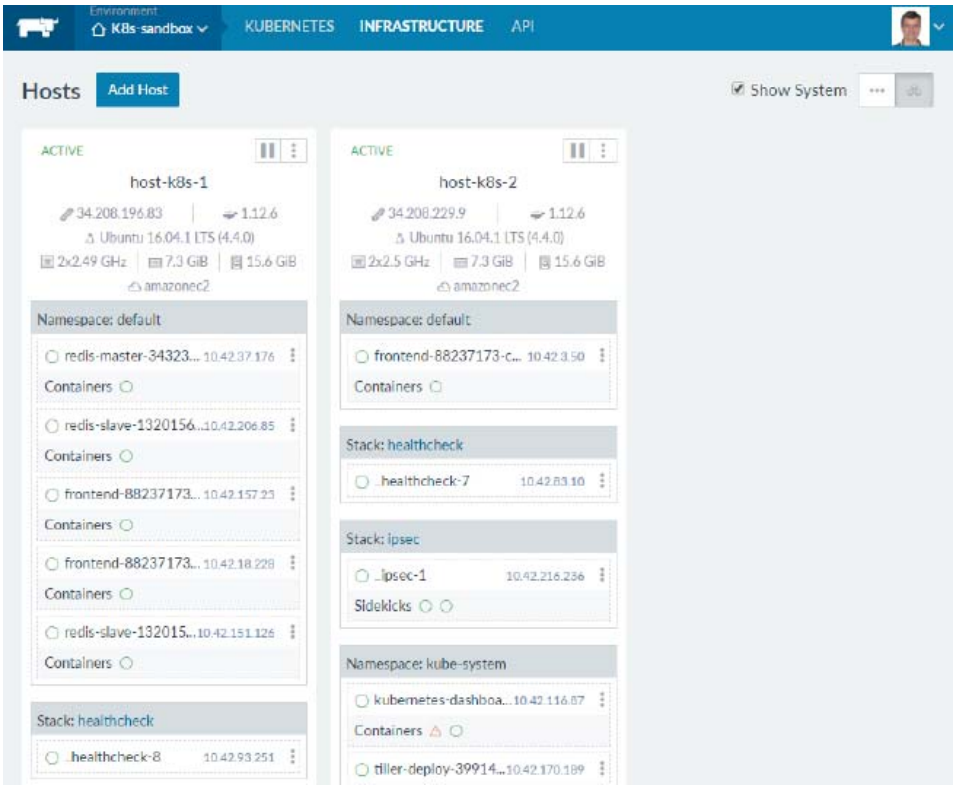

Figure 8: Infrastructure Hosts view in Rancher following automated Kubernetes installation

Rancher provides its own CNI compliant IPsec network and through the Rancher UI users can install application stacks from its own catalog, provision applications using the Kubernetes Dashboard or the *kubectl* command line drawing containers from various public registries. Rancher also automatically installs and configures *Heapster* and *Grafana*, tools used to monitor the Kubernetes environment.

Rancher provides a native application catalog that makes it easy to automate the deployment of containerized application stacks regardless of the orchestration engine chosen. This is useful for organizations that for security or other reasons would prefer to have their own catalog rather than being dependent on public repositories. Rancher provides other capabilities as well including role-based access control (making it easier to manage multi-tenant clusters), high-availability and 7x24 commercial support.

Rancher brings many additional capabilities regardless of the orchestration engine used that can make it easier to deploy and manage clusters and containerized applications over their lifecycle.

Both Docker Swarm and Kubernetes are capable orchestrations. The right choice will depend on a customer's specific needs

SUMMARY

Both Swarm and Kubernetes are highly capable orchestration solutions for Docker containers. While each solution has strengths and weaknesses, the two solutions have broadly similar feature sets.

For users deploying applications with Rancher, the need to decide is less pressing, because users can deploy both orchestration environments concurrently and experiment to decide what solution best meets their needs. Also, Rancher mitigates many of the challenges that new users often experience with Kubernetes by making clusters and containerized applications easier to deploy and manage.

While there is no right or wrong answer, the table below can be helpful in deciding what solution to use when:

Consider Swarm when:	Consider Kubernetes when:
• You're comfortable with Docker and want to get started quickly in a familiar environment • You have existing docker-compose files that you want to use in a clustered environment • You have minimal support resources to dedicate to cluster management and need a simple solution • You are running relatively simple applications and don't anticipate needing the more advanced features found in Kubernetes	• You're prepared to learn a new command line environment and application definition format if it will provide additional functionality • You're running a large, multi-tenant environment and anticipate needing features like namespaces, SLAs and more sophisticated policy controls • You anticipate the need for different types of workloads including batch-oriented workflows, Cron Jobs or Stateful services with special requirements • You have dedicated cluster administrators, expect to scale, and see value in fine tuning policies to support individual application workloads

About the author: *Gord Sissons is a consultant at StoryTek Consulting Inc. located near Toronto, Ontario, Canada. His backgound includes development, product management, marketing and consulting. Gord is a frequent contributor to Rancher blog and works with a number of technology companies in HPC, Big Data and cluster management technologies.*

You can follow Gord on Twitter (@GJSissons) or connect with him on LinkedIn at https://www.linkedin.com/in/gordsissons/.

APPENDIX - DETAILED FEATURE COMPARISON

The table below provides detailed feature level comparison between Docker Swarm and Kubernetes. It should be noted that just because one of the orchestrators lacks a specific feature is not always a bad thing, since it may take a different approach to solving the same problem. Also, when deployed with Rancher, many of the gaps identified below are mitigated.

Feature	Docker Swarm		Kubernetes	
Architectural Features				
Sharing state between masters	Yes	internal RAFT algorithm	Yes	RAFT, external etcd cluster
Support for replica sets	No	Implicit, but not exposed	Yes	Replica sets exposed
Support for services	Yes	Services exposed in Swarm	Yes	Yes, comprised of Pods
Multi-host application stacks	Yes	Stacks new in Swarm	Yes	Kubernetes deployments
Support for load balancers	Yes	Implicit in services	Yes	Yes, multiple solutions
Health checks	Yes	Basic healthchecks in YAML	Yes	Liveness and Readiness probes
Application specifications	Yes	via Docker Compose, DAB	Yes	Native YAML specification
Security	Yes	TLS mutual authentication	Yes	Requires effort to setup
Command Interfaces				
Command line interface	Yes	Docker command set	Yes	Kubectl CLI
Docker command set compatibility	Yes	Uses Docker command set	No	Separate command set
REST API	Yes	Native Docker API	Yes	Kubernetes API
Web console	Yes	Third party	Yes	Native Kubernetes Dashboard
Networking Features				
Automated network configuration	Yes	Included in Docker	No	Not handled by Kubernetes
Overlay networks / SDN	Yes	Native overlay network	Yes	Multiple networks supported
Flat network model	No	Overlays manually created	Yes	All Pods reachable by default
Routing mesh support	Yes	Route from any node	Yes	Using NodePort functionality
Ingress support	No	Not required by design	Yes	Configurable Ingress resource
Storage Features				
Persistent storage	Yes	Using plug-ins	Yes	Many volume types
Mount multiple volumes per container	No	One volume per container	Yes	Support for multiple volumes
Secret management				
Native secret management facility	Yes	Native in swam mode only	Yes	Native in Kubernetes
Pass secrets via filesystem	Yes	In-memory directory path	Yes	Mount secrets in volumes
Pass secrets via environment variables	No	No functionality	Yes	Optionally pass in ENV vars.
Scheduling Features				
Job support (run to completion)	No	Not supported	Yes	Yes, batch Jobs
Pet Sets functionality	No	Not supported	Yes	Now called Stateful Sets
Schedule jobs / Cron jobs	No	Not supported	Yes	Cron Jobs
Affinity / Anti-Affinity	No	In standalone SWARM only	Yes	Yes, advanced policies
Resource reservations / limits	Yes	Yes, in docker compose	Yes	Yes, in YAML specification
Constraint based scheduling	Yes	Supported	Yes	Supported
Scheduling performance	Good	Good response time	Good	Slower than Swarm

Feature	Docker Swarm		Kubernetes	
Scalability				
Manually scale services up and down	Yes	docker service scale	Yes	Scale services / replicas
Auto-scale based on policy	No	No capability	Yes	Autoscale feature
Tested at scale	Yes	1,000 nodes, 50,000 cont.	Yes	5,000 nodes in K8s 1.6
Namespace support	No	No such concept	Yes	Segment apps by namespace
Container Lifecycle Hooks	No	No such concept	Yes	PostStart, PreStop
Guaranteed Scheduling	No	No such concept	Yes	Evicts less critical Pods
Management Features				
Dynamically add nodes to cluster	Yes	Docker swarm add	No	Handled by 3rd party mgrs.
Easily change node role at runtime	Yes	Docker swarm leave/join	No	No, requires manual effort
Native Centralized Logs	Yes	Docker service logs	No	Relies on fluentd, others
Security Scanning	Yes	Docker EE advanced	No	Third party solutions
Vulnerability Monitoring	Yes	Docker EE advanced	No	Third party solutions
Support for Draining Nodes	Yes	Basic feature	Yes	Basic feature

REFERENCES

1. Evaluating container platforms at scale - https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c
2. The scalability test GitHub source code available at https://blog.docker.com/2015/11/scale-testing-docker-swarm-30000-containers/
3. Comparative swarm / Kubernetes benchmark https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c
4. Scalability in Kubernetes 1.6 - http://blog.kubernetes.io/2017/03/scalability-updates-in-kubernetes-1.6.html
5. Many deployment options for Kubernetes - https://kubernetes.io/docs/setup/pick-right-solution/
6. Container Networking Initiative - https://github.com/containernetworking/cni
7. Kubernetes performance - http://blog.kubernetes.io/2017/03/scalability-updates-in-kubernetes-1.6.html
8. Updates to Kubernetes Performance in 1.3 - http://blog.kubernetes.io/2017/03/scalability-updates-in-kubernetes-1.6.html
9. Comparative Swarm, Kubernetes benchmark - https://medium.com/on-docker/evaluating-container-platforms-at-scale-5e7b44d93f2c