Swarm v. Fleet v. Kubernetes v. Mesos

Comparing different orchestration tools.

By Adrian Mouat, October 21, 2015

Dublin Philharmonic Orchestra performing in North Carolina (source: Wikimedia Commoms).

Take the Ops survey. Help us learn about the demographics, work environments, tools, and compensation of practitioners in our growing field.

Most software systems evolve over time. New features are added and old ones pruned. Fluctuating user demand means an efficient system must be able to quickly scale resources up and down. Demands for near zero-downtime require automatic fail-over to pre-provisioned back-up systems, normally in a separate data centre or region.

On top of this, organizations often have multiple such systems to run, or need to run occasional tasks such as data-mining that are separate from the main system, but require significant resources or talk to the existing system.

When using multiple resources, it is important to make sure they are efficiently used — not sitting idle — but can still cope with spikes in demand. Balancing cost-effectiveness against the ability to quickly scale is difficult task that can be approached in a variety of ways.

Get O'Reilly's weekly Systems Engineering and Operations newsletter

O'REILLY

Systems Engineering and Operations Newsletter

Top 8 Systems Engineering and Operations Trends for 2017

Forecasting trends is tricky, especially in the fast-moving world of systems operations and engineering. This year, at our Velocity Conference, we have talked about distributed systems, SRE, containerization, serverless architectures, burnout, and many other topics related to the human and technological challenges of delivering software. Here are some of the trends we see for the next year:

1. Distributed Systems

We think this is important enough that we re-focused the entire Velocity conference on it (and **renamed this newsletter to reflect that**).

Email Address Subscribe

We protect your privacy.

All of this means that the running a non-trivial system is full of administrative tasks and challenges, the complexity of which should not be underestimated. It quickly becomes impossible to look after machines on an individual level; rather than patching and updating machines one-by-one they must be treated identically. When a machine develops a problem it should be destroyed and replaced, rather than nursed back to health.

Various software tools and solutions exist to help with these challenges. Let's focus on orchestration tools, which help make all the pieces work together, working with the cluster to start containers on appropriate hosts and connect them together. Along the way, we'll consider scaling and automatic failover, which are important features.

Swarm

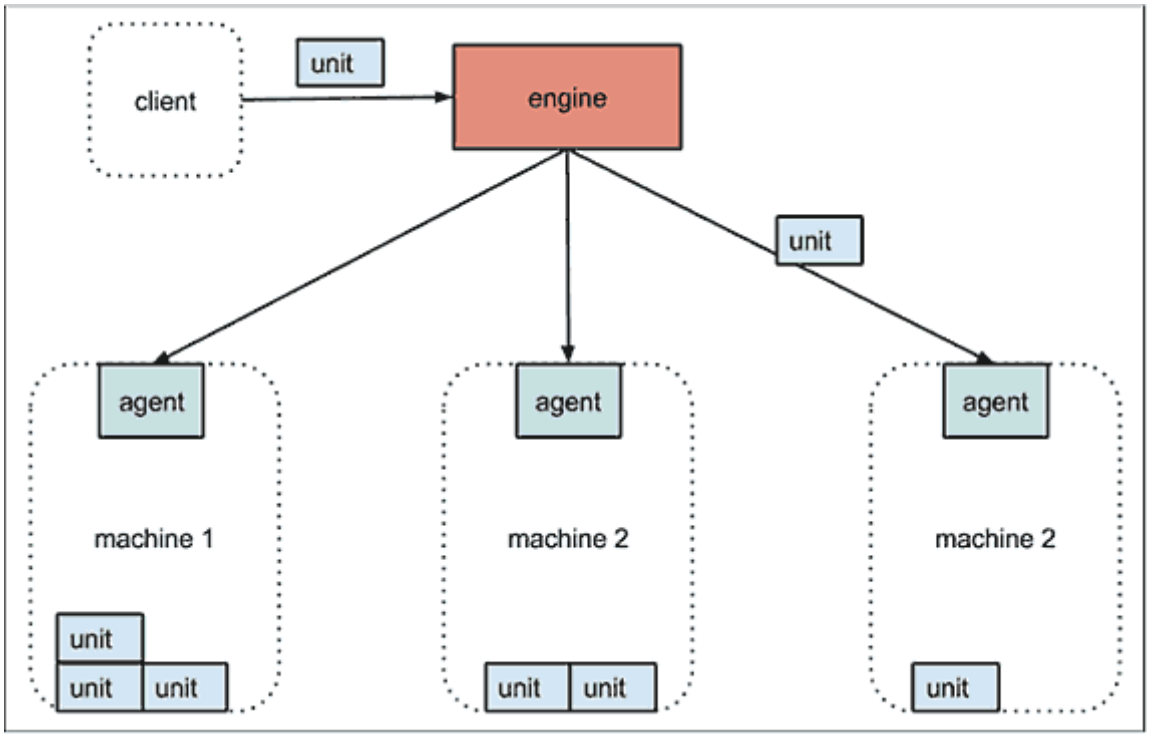
Swarm is the native clustering tool for Docker. Swarm uses the standard Docker API, meaning containers can be launched using normal `docker run` commands and Swarm will take care of selecting an appropriate host to run the container on. This also means that other tools which use the Docker API — such as Compose and bespoke scripts — can use Swarm without any changes and take advantage of running on a cluster rather than a single host.

The basic architecture of Swarm is fairly straightforward: each host runs a Swarm *agent* and one host runs a Swarm *manager* (on small test clusters this host may also run an agent). The manager is responsible for the orchestration and scheduling of containers on the hosts. Swarm can be run in a high-availability mode where one of etcd, Consul or ZooKeeper is used to handle fail-over to a back-up manager. There are several different methods for how hosts are found and added to a cluster, which is known as *discovery* in Swarm. By default, *token* based discovery is used, where the addresses of hosts are kept in a list stored on the Docker Hub.

Fleet

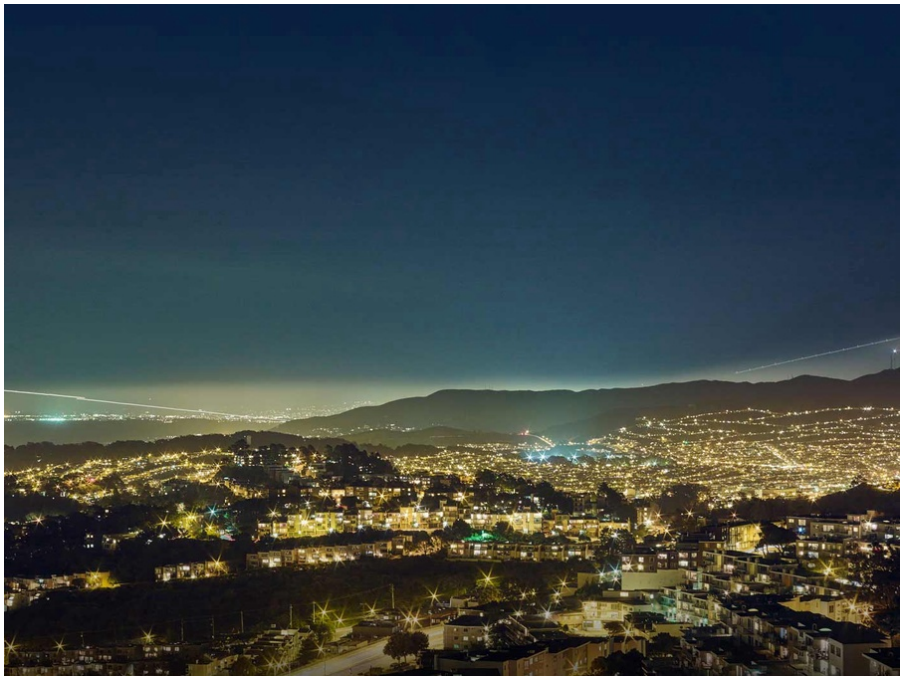
Fleet is the cluster management tool from CoreOS. It bills itself as a "low-level cluster engine", meaning that it is expected to form a "foundation layer" for higher-level solutions such as Kubernetes.

The most distinguishing feature of fleet is that it builds on top of systemd. Whereas systemd provides system and service initialization for a single machine, fleet extends this to a cluster of machines. Fleet reads systemd unit files, which are then scheduled on a machine or machines in the cluster.



The technical architecture of fleet is shown in Figure 12-2. Each machine runs an *engine* and an *agent*. Only one engine is active in the cluster at any time, but all agents are constantly running (for the sake of the diagram, the active engine is shown separately to the machines, but it will be running on one of them). Systemd unit files (henceforth *units*) are submitted to the engine, which will schedule the job on the "least-loaded" machine. The unit file will normally simply run a container. The agent takes care of starting the unit and reporting state. Etcd is used to enable communication between machines and store the status of the cluster and units.

O'REILLY VELOCITY CONFERENCE



Velocity San Jose 2017, June 19-22

Learn More

The architecture is designed to be fault-tolerant; if a machine dies, any units scheduled on that machine will be restarted on new hosts.

Fleet supports various scheduling hints and constraints. At the most basic level, units can be scheduled as *global*, meaning an instance will run on all machines, or as a single unit which will run on a single machine. Global scheduling is very useful for utility containers for tasks such as logging and monitoring. Various *affinity* type constraints are supported, so for example a container that runs a health check can be scheduled to always run next to the application server. Metadata can also be attached to hosts and used for scheduling, so you could for example ask for your containers to run on machines belonging to a given region or with certain hardware.

As fleet is based on systemd, it also supports the concept of *socket activation*; a container can be spun up in response to a connection on a given port. The primary advantage of this is that processes can be created just-in-time, rather than sitting around idle waiting for something to happen. There are potentially other benefits related to management of sockets, such as not losing messages between container restarts.

Kubernetes

Kubernetes is a container orchestration tool built by Google, based on their experiences using containers in production over the last decade. Kubernetes is somewhat opinionated and enforces several concepts around how containers are organized and networked. The primary concepts you need to understand are:

- ***Pods*** – Pods are groups of containers that are deployed and scheduled together. Pods form the atomic unit of scheduling in Kubernetes, as opposed to single containers in other systems. A pod will typically include 1 to 5 containers which work together to provide a service. In addition to these user containers, Kubernetes will run other containers to provide logging and monitoring services. Pods are treated as ephemeral in Kubernetes; you should expect them to be created and destoyed continually as the system evolves.
- ***Flat Networking Space*** – Networking is very different in Kubernetes to the default Docker networking. In the default Docker networking, containers live on a private subnet and can't communicate directly with containers on other hosts without forwarding ports on the host or using proxies. In Kubernetes, containers within a pod share an IP address, but the address space is "flat" across all pods, meaning all pods can talk to each other without any Network Address Translation (NAT). This makes multi-host clusters much more easy to manage, at the cost of not supporting links and making single host (or, more accurately, single pod) networking a little more tricky. As containers in the same pod share an IP, they can communicate by using ports on the localhost address (which does mean you need to coordinate port usage within a pod).
- ***Labels*** – Labels are key-value pairs attached to objects in Kubernetes, primarily pods, used to describe identifying characteristics of the object e.g. version: dev and tier: frontend. Labels are not normally unique; they are expected to identify groups of containers. Label selectors can then be used to identify objects or groups of objects, for example all the pods in the frontend tier with environment set to production. Through using labels, it is easy to do grouping tasks such as assigning pods to load-balanced groups or moving pods between groups.
- ***Services*** – Services are stable endpoints that can be addressed by name. Services can be connected to pods by using label selectors; for example my "cache" service may connect to several "redis" pods identified by the label selector "type": "redis". The service will automatically round-robin requests between the pods. In this way, services can be used to connect parts of a system to each other. Using services provides a layer of abstraction that means applications do not need to know internal details of the

service they are calling; for example application code running inside a pod only needs to know the name and port of the database service to call, it does not care how many pods make up the database, or which pod it talked to last time. Kubernetes will set up a DNS server for the cluster that watches for new services and allows them to be addressed by name in application code and configuration files.
It is also possible to set up services which do not point to pods but to other preexisting services such as external APIs or databases.

- **Replication Controllers** – Replication controllers are the normal way to instantiate pods in Kubernetes (typically, you don't use the Docker CLI in Kubernetes). They control and monitor the number of running pods (called replicas) for a service. For example, a replication controller may be responsible for keeping 5 Redis pods running. Should one fail, it will immediately launch a new one. If the number of replicas is reduced, it will stop any excess pods. Although using replication controllers to instantiate all pods adds an extra layer of configuration, it also significantly improves fault tolerance and reliability.

Mesos and Marathon

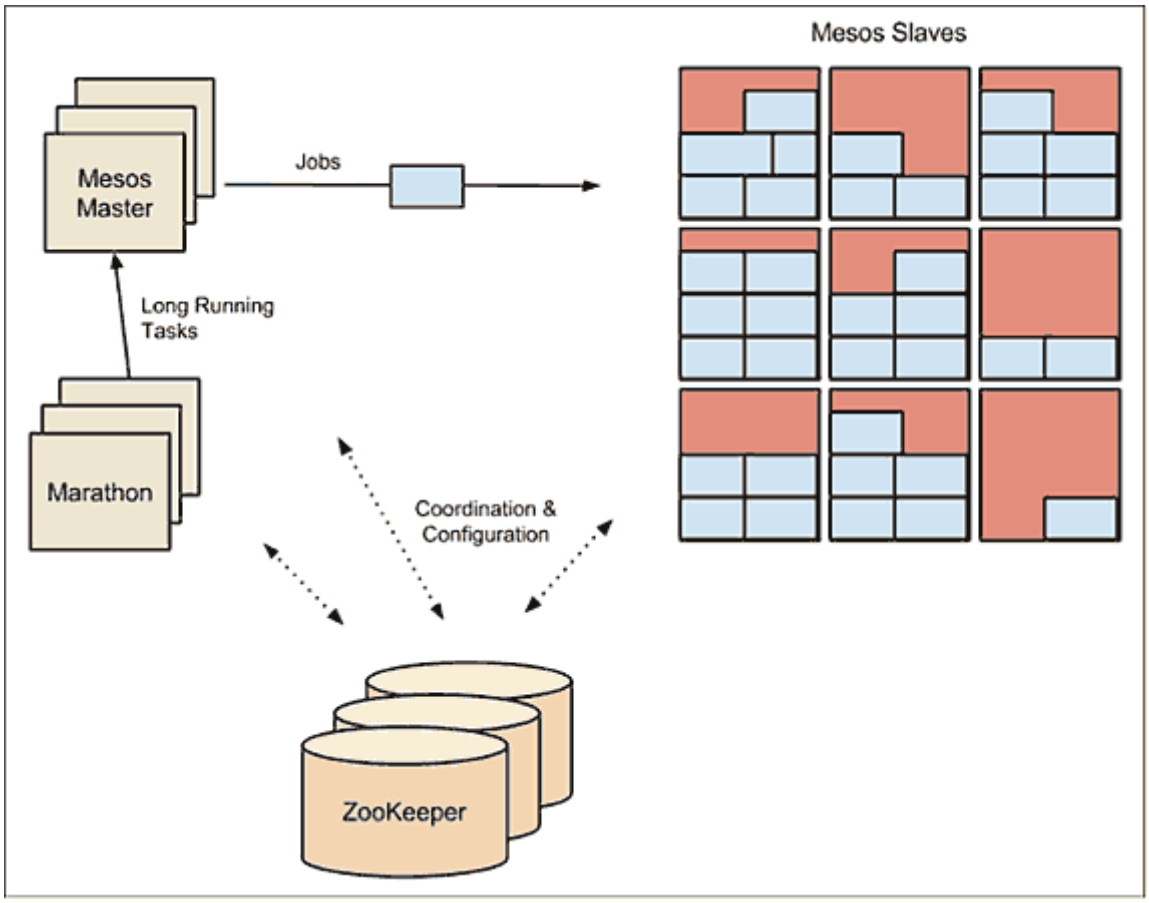
Apache Mesos (https://mesos.apache.org) is an open-source cluster manager. It's designed to scale to very large clusters involving hundreds or thousands of hosts. Mesos supports diverse workloads from multiple tenants; one user's Docker containers may be running next to another user's Hadoop tasks.

Apache Mesos was started as a project at the University of Berkeley before becoming the underlying infrastructure used to power Twitter and an important tool at many major companies such as eBay and Airbnb. A lot of continuing development in Mesos and supporting tools (such as Marathon) is undertaken by Mesosphere, a company co-founded by Ben Hindman, one of the orginal developers of Mesos.

The architecture of Mesos is designed around high-availability and resilience. The major components in a Mesos cluster are:

- ***Mesos Agent Nodes*** – Responsible for actually running tasks. All agents submit a list of their available resources to the master. There will typically be 10s to 1000s of agent nodes.

- ***Mesos Master*** – The master is responsible for sending tasks to the agents. It maintains a list of available resources and makes "offers" of them to frameworks. The master decides how many resources to offer based on an allocation strategy. There will typically be 2 or 4 stand-by masters ready to take over in case of a failure.
- ***ZooKeeper*** – Used in elections and for looking up address of current master. Typically 3 or 5 ZooKeeper instances will be running to ensure availability and handle failures.
- ***Frameworks*** – Frameworks co-ordinate with the master to schedule tasks onto agent nodes. Frameworks are composed of two parts; the *executor* process which runs on the agents and takes care of running the tasks and the *scheduler* which registers with the master and selects which resources to use based on offers from the master. There may be multiple frameworks running on a Mesos cluster for different kinds of task. Users wishing to submit jobs interact with frameworks rather than directly with Mesos.



In Figure 12-4 we see a Mesos cluster which uses the Marathon framework as the scheduler. The Marathon scheduler uses ZooKeeper to locate the current Mesos master which it will submit tasks to. Both the Marathon scheduler and the Mesos master have stand-bys ready to start work should the current master become unavailable.

Typically, ZooKeeper will run on the same hosts as the Mesos master and its standbys. In a small cluster, these hosts may also run agents, but larger clusters require communication with the master, making this less feasible. Marathon may run on the same hosts as well, or may instead run on separate hosts which live on the network boundary and form the access point for clients, thus keeping clients separated from the Mesos cluster itself.

Marathon (from Mesosphere) is designed to start, monitor and scale long-running applications. Marathon is designed to be flexible about the applications it launches, and can even be used to start other complementary frameworks such Chronos ("cron" for the datacenter). It makes a good choice of framework for running Docker containers, which are directly supported in Marathon. Like the other orchestration frameworks we've looked at, Marathon supports various affinity and constraint rules. Clients interact with Marathon through a REST API. Other features include support for health checks and an event stream that can be used to integrate with load-balancers or for analyzing metrics.

Conclusion

There are clearly a lot of choices for orchestrating, clustering, and managing containers. That being said, the choices are generally well differentiated. In terms of orchestration, we can say the following:

- Swarm has the advantage (and disadvantage) of using the standard Docker interface. Whilst this makes it very simple to use Swarm and to integrate it into existing workflows, it may also make it more difficult to support the more complex scheduling that may be defined in custom interfaces.
- Fleet is a low-level and fairly simple orchestration layer that can be used as a base for running higher level orchestration tools, such as Kubernetes or custom systems.
- Kubernetes is an opinionated orchestration tool that comes with service discovery and replication baked-in. It may require some re-designing of existing applications, but used correctly will result in a fault-tolerant and scalable system.
- Mesos is a low-level, battle-hardened scheduler that supports several frameworks for container orchestration including Marathon, Kubernetes, and Swarm. At the time of writing, Kubernetes and Mesos are more developed and stable than Swarm. In terms of scale, only Mesos has been proven to support large-scale systems of hundreds or

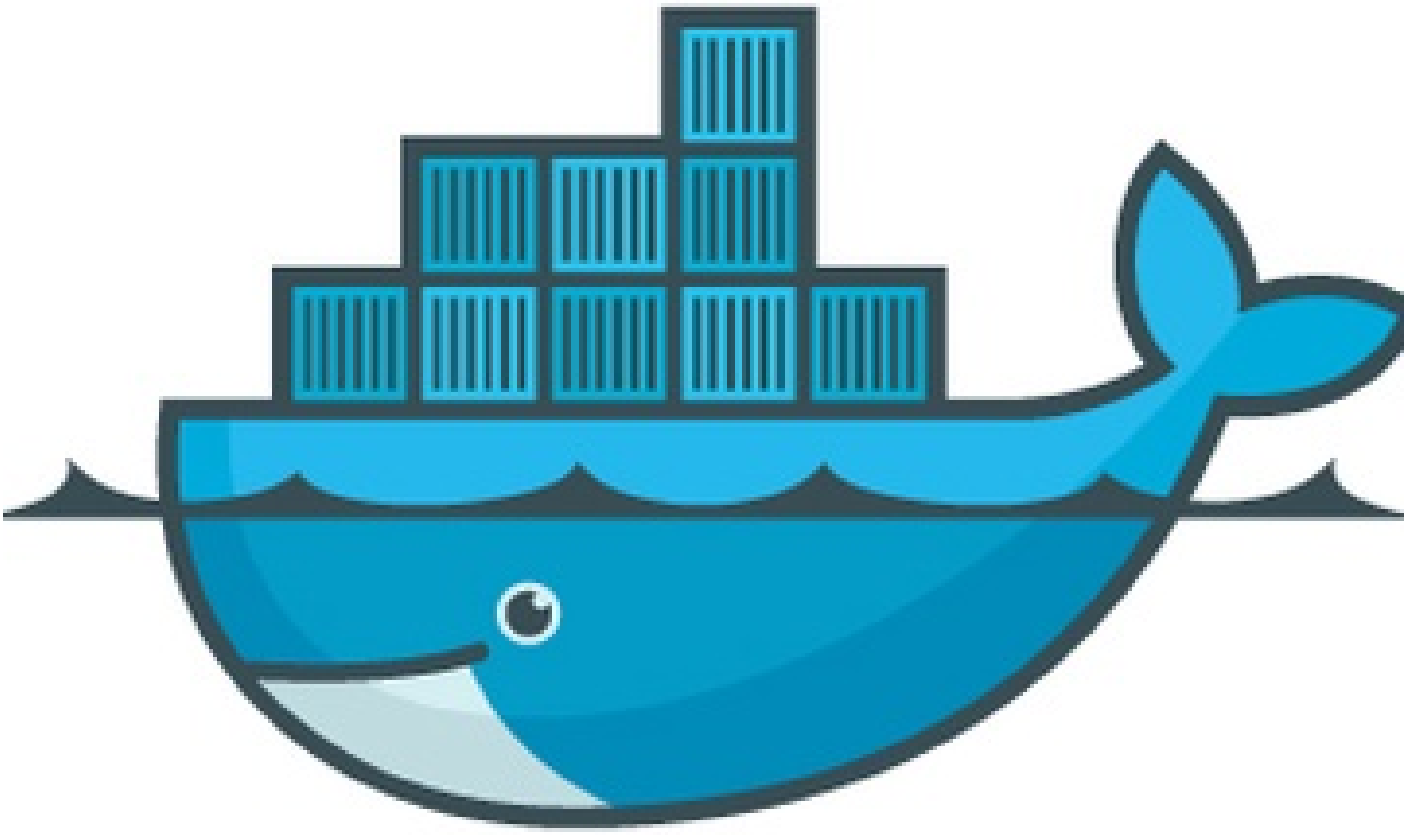
thousands of nodes. However, when looking at small clusters of, say, less than a dozen nodes, Mesos may be an overly complex solution.

Article image: Dublin Philharmonic Orchestra performing in North Carolina (source: Wikimedia Commoms).

Adrian Mouat

Adrian Mouat is Chief Scientist at Container Solutions. In the past he has worked on a wide range of software projects, from small webapps to large scale data-analysis software.

INFRASTRUCTURE



10 things you need to know about Docker

By Andrew Baker

What you should know before before you jump on the container bandwagon.

INFRASTRUCTURE

Practical network automation using Python and Ansible

By Kirk Byers

Learn how to use Python and Ansible to automate networking tasks.

INFRASTRUCTURE

The cloud-native future

By Casey West

Moving beyond ad-hoc automation to take advantage of patterns that deliver predictable capabilities.

INFRASTRUCTURE

An introduction to immutable infrastructure

By Josh Stella

Why you should stop managing infrastructure and start really programming it.

ABOUT US

Our Company

Work with Us

Customer Service

Contact Us

SITE MAP

Ideas

Learning

Topics

All

© 2017 O'Reilly Media, Inc. All trademarks and registered trademarks appearing on oreilly.com are the property of their respective owners.

Terms of Service • **Privacy Policy** • **Editorial Independence**