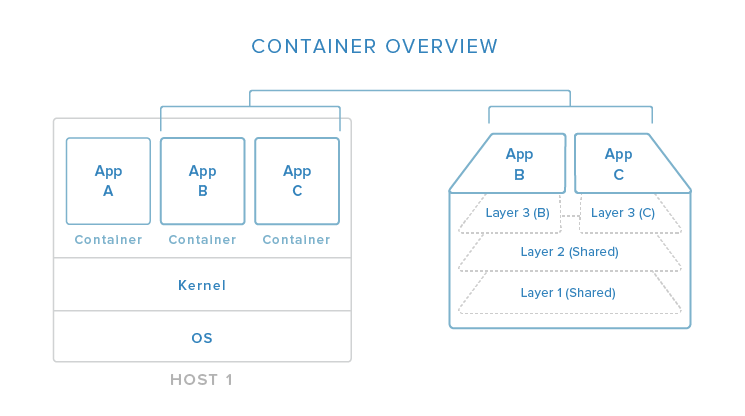
[Docker](https://www.docker.com/) is an application that simplifies the process of managing application processes in containers. Containers let you run your applications in resource-isolated processes. They’re similar to virtual machines, but containers are more portable, more resource-friendly, and more dependent on the host operating system.

[**Introduction**](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-introduction-to-common-components#introduction)

Containerization is the process of distributing and deploying applications in a portable and predictable way. It accomplishes this by packaging components and their dependencies into standardized, isolated, lightweight process environments called containers.  Many organizations are now interested in designing applications and services that can be easily deployed to distributed systems, allowing the system to scale easily and survive machine and application failures.  Docker, a containerization platform developed to simplify and standardize deployment in various environments, was largely instrumental in spurring the adoption of this style of service design and management.  A large amount of software has been created to build on this ecosystem of distributed container management.

[**Docker and Containerization**](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-introduction-to-common-components#docker-and-containerization)

Docker is the most common containerization software in use today. While other containerizing systems exist, Docker makes container creation and management simple and integrates with many open source projects.



In this image, you can begin to see (in a simplified view) how containers relate to the host system. Containers isolate individual applications and use operating system resources that have been abstracted by Docker. In the exploded view on the right, we can see that containers can be built by “layering”, with multiple containers sharing underlying layers, decreasing resource usage.

Docker’s main advantages are:

* **Lightweight resource utilization**: instead of virtualizing an entire operating system, containers isolate at the process level and use the host’s kernel.
* **Portability**: all of the dependencies for a containerized application are bundled inside of the container, allowing it to run on any Docker host.
* **Predictability**: The host does not care about what is running inside of the container and the container does not care about which host it is running on.  The interfaces are standardized and the interactions are predictable.

Typically, when designing an application or service to use Docker, it works best to break out functionality into individual containers, a design decision known as service-oriented architecture. This gives you the ability to easily scale or update components independently in the future. Having this flexibility is one of the many reasons that people are interested in Docker for development and deployment.

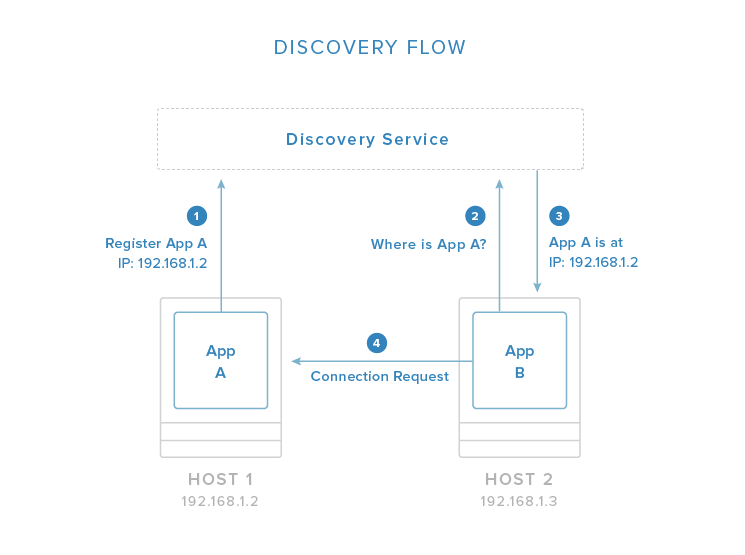
To find out more about containerizing applications with Docker, click [here](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-overview-of-containerization).

[**Service Discovery and Global Configuration Stores**](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-introduction-to-common-components#service-discovery-and-global-configuration-stores)

Service discovery is one component of an overall strategy aimed at making container deployments scalable and flexible. Service discovery is used so that containers can find out about the environment they have been introduced to without administrator intervention. They can find connection information for the components they must interact with, and they can register themselves so that other tools know that they are available. These tools also typically function as globally distributed configuration stores where arbitrary config settings can be set for the services operating in your infrastructure.

A screenshot of a computer program

Description automatically generated



In the above image, you can see an example flow in which one application registers its connection information with the discovery service system. Once registered, other applications can query the discovery service to find out how to connect to the application.

These tools are often implemented as simple key-value stores that are distributed among the hosts in a clustered environment. Generally, the key-value stores provide an HTTP API for accessing and setting values. Some include additional security measures like encrypted entries or access control mechanisms. The distributed stores are essential for managing the clustered Docker hosts in addition to their primary function of providing self-configuration details for new containers.

Some of the responsibilities of service discovery stores are:

* Allowing applications to obtain the data needed to connect with to the services they depend on.
* Allowing services to register their connection information for the above purpose.
* Providing a globally accessible location to store arbitrary configuration data.
* Storing information about cluster members as needed by any cluster management software.

Some popular service discovery tools and related projects are:

* [**etcd**](https://www.digitalocean.com/community/tutorials/how-to-use-etcdctl-and-etcd-coreos-s-distributed-key-value-store): service discovery / globally distributed key-value store
* [**consul**](https://www.digitalocean.com/community/tutorials/an-introduction-to-using-consul-a-service-discovery-system-on-ubuntu-14-04): service discovery / globally distributed key-value store
* [**zookeeper**](https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere#a-basic-overview-of-apache-mesos): service discovery / globally distributed key-value store
* [**crypt**](http://xordataexchange.github.io/crypt/): project to encrypt etcd entries
* [**confd**](https://www.digitalocean.com/community/tutorials/how-to-use-confd-and-etcd-to-dynamically-reconfigure-services-in-coreos): watches key-value store for changes and triggers reconfiguration of services with new values

To learn more about service discovery with Docker, visit our guide [here](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-service-discovery-and-distributed-configuration-stores).

[**Networking Tools**](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-introduction-to-common-components#networking-tools)

Containerized applications lend themselves to a service-oriented design that encourages breaking out functionality into discrete components. While this makes management and scaling easier, it requires even more assurance regarding the functionality and reliability of networking between the components. Docker itself provides the basic networking structures necessary for container-to-container and container-to-host communication.

Docker’s native networking capabilities provide two mechanisms for hooking containers together. The first is to expose a container’s ports and optionally map to the host system for external routing. You can select the host port to map to or allow Docker to randomly choose a high, unused port. This is a generic way of providing access to a container that works well for most purposes.

The other method is to allow containers to communicate by using Docker “links”. A linked container will get connection information about its counterpart, allowing it to automatically connect if it is configured to pay attention to those variables. This allows contact between containers on the same host without having to know beforehand the port or address where the service will be located.

This basic level of networking is suitable for single-host or closely managed environments. However, the Docker ecosystem has produce a variety of projects that focus on expanding the networking functionality available to operators and developers. Some additional networking capabilities available through additional tools include:

* Overlay networking to simplify and unify the address space across multiple hosts.
* Virtual private networks adapted to provide secure communication between various components.
* Assigning per-host or per-application subnetting
* Establishing macvlan interfaces for communication
* Configuring custom MAC addresses, gateways, etc. for your containers

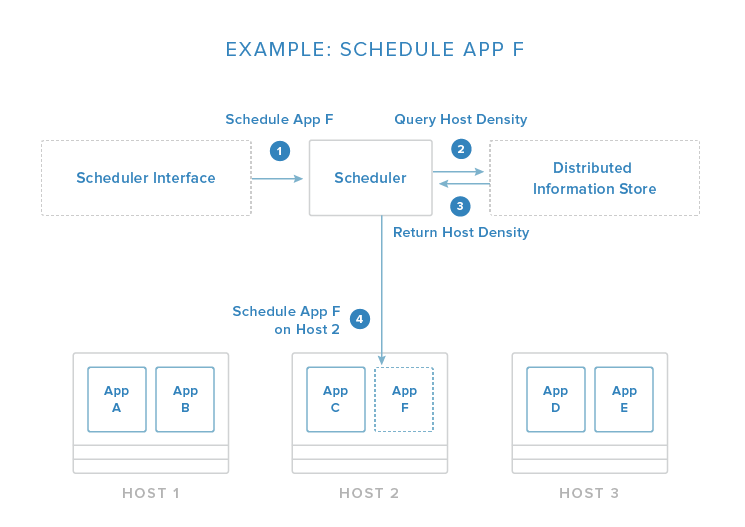
Some projects that are involved with improving Docker networking are:

* **flannel**: Overlay network providing each host with a separate subnet.
* **weave**: Overlay network portraying all containers on a single network.
* **pipework**: Advanced networking toolkit for arbitrarily advanced networking configurations.

For a more in-depth look at the different approaches to networking with Docker, click [here](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-networking-and-communication).

[**Scheduling, Cluster Management, and Orchestration**](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-introduction-to-common-components#scheduling-cluster-management-and-orchestration)

Another component needed when building a clustered container environment is a scheduler. Schedulers are responsible for starting containers on the available hosts.



The image above demonstrates a simplified scheduling decision. The request is given through an API or management tool. From here, the scheduler evaluates the conditions of the request and the state of the available hosts. In this example, it pulls information about container density from a distributed data store / discovery service (as discussed above) so that it can place the new application on the least busy host.

This host selection process is one of the core responsibilities of the scheduler. Usually, it has functions that automate this process with the administrator having the option to specify certain constraints. Some of these constraints may be:

* Schedule the container on the same host as another given container.
* Make sure that the container is not placed on the same host as another given container.
* Place the container on a host with a matching label or metadata.
* Place the container on the least busy host.
* Run the container on every host in the cluster.

The scheduler is responsible for loading containers onto relevant hosts and starting, stopping, and managing the life cycle of the process.

Because the scheduler must interact with each host in the group, cluster management functions are also typically included. These allow the scheduler to get information about the members and perform administration tasks. Orchestration in this context generally refers to the combination of container scheduling and managing hosts.

Some popular projects that function as schedulers and fleet management tools are:

* [**fleet**](https://www.digitalocean.com/community/tutorials/how-to-use-fleet-and-fleetctl-to-manage-your-coreos-cluster): scheduler and cluster management tool.
* [**marathon**](https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere#a-basic-overview-of-marathon): scheduler and service management tool.
* [**Swarm**](https://github.com/docker/swarm/): scheduler and service management tool.
* [**mesos**](https://www.digitalocean.com/community/tutorials/an-introduction-to-mesosphere#a-basic-overview-of-apache-mesos): host abstraction service that consolidates host resources for the scheduler.
* [**kubernetes**](https://www.digitalocean.com/community/tutorials/an-introduction-to-kubernetes): advanced scheduler capable of managing container groups.
* [**compose**](https://github.com/docker/docker/issues/9694): container orchestration tool for creating container groups.

To find out more about basic scheduling, container grouping, and cluster management software for Docker, click [here](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-scheduling-and-orchestration).

[**Conclusion**](https://www.digitalocean.com/community/tutorials/the-docker-ecosystem-an-introduction-to-common-components#conclusion)

By now, you should be familiar with the general function of most of the software associated with the Docker ecosystem. Docker itself, along with all of the supporting projects, provide a software management, design, and deployment strategy that enables massive scalability. By understanding and leveraging the capabilities of various projects, you can execute complex application deployments that are flexible enough to account for variable operating requirements.

A screenshot of a computer code

Description automatically generated