Applications requirements and networking environments are diverse and sometimes opposing forces. In between applications and the network sits Docker networking, affectionately called the Container Network Model (https://github.com/docker/libnetwork/blob/master/docs/design.md) or CNM. It's CNM that brokers connectivity for your Docker containers and also what abstracts away the diversity and complexity so common in networking. The result is portability and it comes from CNM's powerful **network drivers**. These are pluggable interfaces for the Docker Engine, Swarm, and UCP that provide special capabilities like multi-host networking, network layer encryption, and service discovery.

Naturally, the next question is ***which network driver should I use****?* Each driver offers tradeoffs and has different advantages depending on the use case. There are **built-in** network drivers that come included with Docker Engine and there are also **plug-in** network drivers offered by networking vendors and the community. The most commonly used built-in network drivers are **bridge**, **overlay** and **macvlan.** Together they cover a very broad list of networking use cases and environments. For a more in depth comparison and discussion of even more network drivers, check out the Network Reference Architecture. (https://success.docker.com/Datacenter/Apply/Docker_Reference_Architecture%3A_Designing_Scalable%2C (https://success.docker.com/Datacenter/Apply/Docker_Reference_Architecture%3A_Designing_Scalable%2C

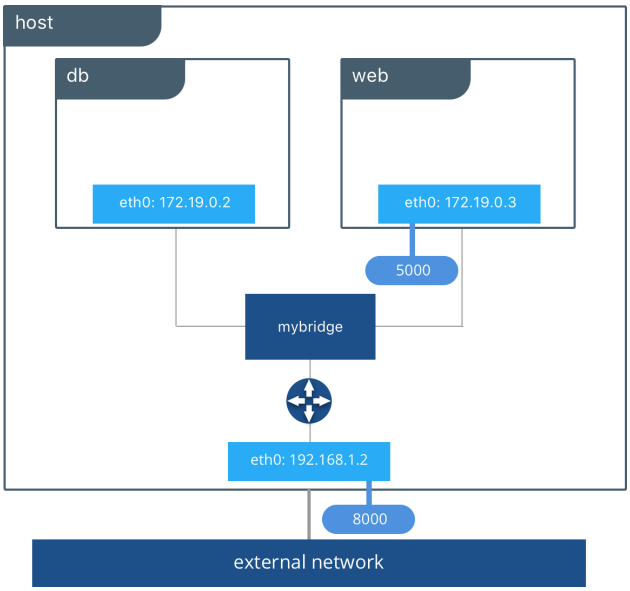
Bridge Network Driver

The `bridge` networking driver is the first driver on our list. It's simple to understand, simple to use, and simple to troubleshoot, which makes it a good networking choice for developers and those new to Docker. The `bridge` driver creates a private network internal to the host so containers on this network can communicate. External access is granted by exposing ports to containers. Docker secures the network by managing rules that block connectivity between different Docker networks.

Behind the scenes, the Docker Engine creates the necessary Linux bridges, internal interfaces, iptables rules, and host routes to make this connectivity possible. In the example highlighted below, a Docker bridge network is created and two containers are attached to it. With no extra configuration the Docker Engine does the necessary wiring, provides service discovery for the containers, and configures security rules to prevent communication to other networks. A built-in IPAM driver provides the container interfaces with private IP addresses from the subnet of the bridge network.

In the following examples, we use a fictitious app called `pets` comprised of a `web` and `db` container. Feel free to try it out on your own UCP or Swarm cluster. Your app will be accessible on `` `<host-ip>:8000`. ``

```
docker network create -d bridge mybridge
docker run -d --net mybridge --name db redis
docker run -d --net mybridge -e DB=db -p 8000:5000 --name web chrch/web
```



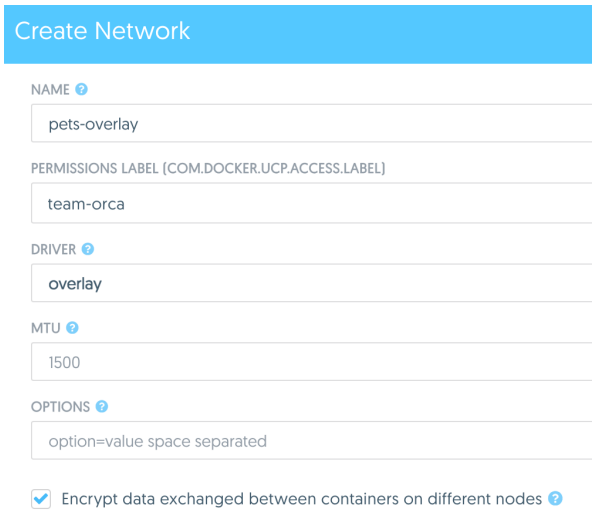
Our application is now being served on our host at port 8000. The Docker bridge is allowing `web` to communicate with `db` by its container name. The bridge driver does the service discovery for us automatically because they are on the same network. All of the port mappings, security rules, and pipework between Linux bridges is handled for us by the networking driver as containers are scheduled and rescheduled across a cluster.

The bridge driver is a **local scope** driver, which means it only provides service discovery, IPAM, and connectivity on a single host. Multi-host service discovery requires an external solution that can map containers to their host location. This is exactly what makes the `overlay` driver so great.

Overlay Network Driver

The built-in Docker `overlay` network driver radically simplifies many of the complexities in multi-host networking. It is a **swarm scope** driver, which means that it operates across an entire Swarm or UCP cluster rather than individual hosts. With the `overlay` driver, multi-host networks are first-class citizens inside Docker without external provisioning or components. IPAM, service discovery, multi-host connectivity, encryption, and load balancing are built right in. For control, the `overlay` driver uses the encrypted Swarm control plane to manage large scale clusters at low convergence times.

The `overlay` driver utilizes an industry-standard VXLAN data plane that decouples the container network from the underlying physical network (the *underlay*). This has the advantage of providing maximum portability across various cloud and on-premises networks. Network policy, visibility, and security is controlled centrally through the Docker Universal Control Plane (UCP).



In this example we create an overlay network in UCP so we can connect our `web` and `db` containers when they are living on different hosts. Native DNS-based service discovery for services & containers within an overlay network will ensure that `web` can resolve to `db` and vice-versa. We turned on encryption so that communication between our containers is secure by default. Furthermore, visibility and use of the network in UCP is restricted by the permissions label we use.

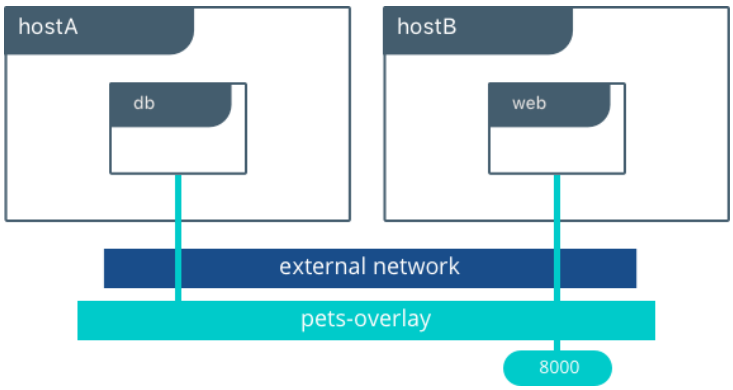
UCP will schedule services across the cluster and UCP will dynamically program the overlay network to provide connectivity to the containers wherever they are. When services are backed by multiple containers, VIP-based load balancing will distribute traffic across all of the containers.

Feel free to run this example against your UCP cluster with the following CLI commands:

```
docker network create -d overlay --opt encrypted pets-overlay

docker service create --network pets-overlay --name db redis

docker service create --network pets-overlay -p 8000:5000 -e DB=db --name web chrch/web
```



In this example we are still serving our web app on port 8000 but now we have deployed our application across different hosts. If we wanted to scale our `web` containers, Swarm & UCP networking would load balance the traffic for us automatically.

The `overlay` driver is a feature-rich driver that handles much of the complexity and integration that organizations struggle with when crafting piecemeal solutions. It provides an out-of-the-box solution for many networking challenges and does so at scale.

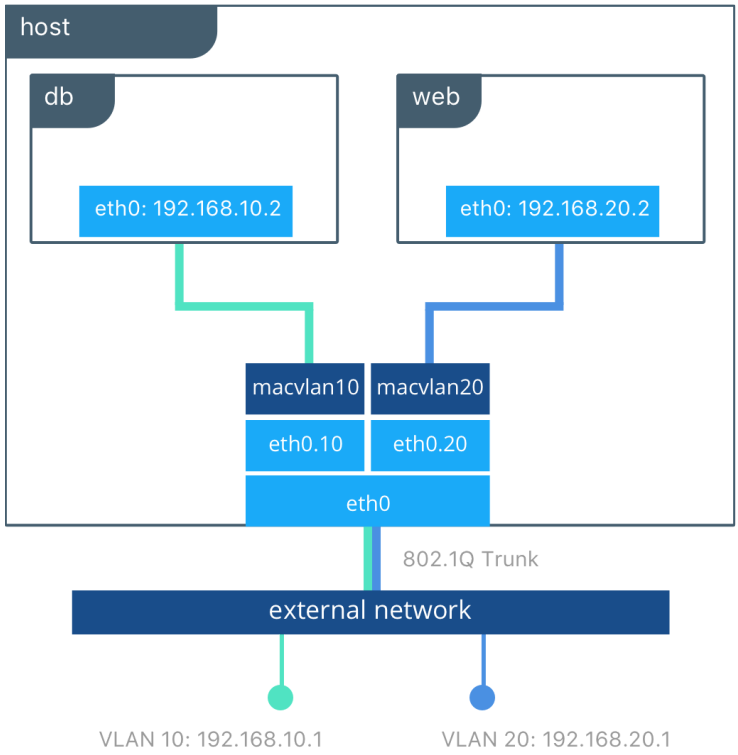
MACVLAN Driver

The `macvlan` driver is the newest built-in network driver and offers several unique characteristics. It's a very lightweight driver, because rather than using any Linux bridging or port mapping, it connects container interfaces directly to host interfaces. Containers are addressed with routable IP addresses that are on the subnet of the external network.

As a result of routable IP addresses, containers communicate directly with resources that exist outside a Swarm cluster without the use of NAT and port mapping. This can aid in network visibility and troubleshooting. Additionally, the direct traffic path between containers and the host interface helps reduce latency. `macvlan` is a local scope network driver which is configured per-host. As a result, there are stricter dependencies between MACVLAN and external networks, which is both a constraint and an advantage that is different from `overlay` or `bridge`.

The `macvlan` driver uses the concept of a parent interface. This interface can be a host interface such as `eth0`, a sub-interface, or even a bonded host adaptor which bundles Ethernet interfaces into a single logical interface. A gateway address from the external network is required during MACVLAN network configuration, as a MACVLAN network is a L2 segment from the container to the network gateway. Like all Docker networks, MACVLAN networks are segmented from each other – providing access within a network, but not between networks.

The `macvlan` driver can be configured in different ways to achieve different results. In the below example we create two MACVLAN networks joined to different subinterfaces. This type of configuration can be used to extend multiple L2 VLANs through the host interface directly to containers. The VLAN default gateway exists in the external network.



The `db` and `web` containers are connected to different MACVLAN networks in this example. Each container resides on its respective external network with an external IP provided from that network. Using this design an operator can control network policy outside of the host and segment containers at L2. The containers could have also been placed in the same VLAN by configuring them on the same MACVLAN network. This just shows the amount of flexibility offered by each network driver.

Portability and choice are important tenants in the Docker philosophy. The Docker Container Network Model provides an open interface for vendors and the community to build network drivers. The complementary evolution of Docker and SDN technologies is providing more options and capabilities every day.

Get familiar with #Docker Network drivers: bridge, overlay, macvlan (https://twitter.com/share?text=Get+familiar+with+%23Docker+Network+drivers%3A+bridge%2C+overlay%2C+macvlan&via=docker&related=docker&url=https://dockr.ly/2hNTDki)

CLICK TO TWEET (HTTPS://TWITTER.COM/SHARE?

TEXT=GET+FAMILIAR+WITH+%23DOCKER+NETWORK+DRIVERS%3A+BRIDGE%2C+OVERLAY%2C+MACVLAN&VIA=DOC

Happy Networking!

More Resources:

- Check out (http://dockr.ly/2fIXpaU) the latest Docker Datacenter networking updates
- Read the latest RA: (https://success.docker.com/?cid=ucp112ra) Docker UCP Service Discovery and Load Balancing
- See What's New in Docker Datacenter (https://blog.docker.com/2016/11/docker-datacenter-adds-enterprise-orchestration-security-policy-refreshed-ui/)
- Sign up (http://www.docker.com/products/docker-datacenter)for a free 30 day trial

docker networking (/tag/docker-networking/), *libnetwork. network drivers (/tag/libnetwork-network-drivers/)*, *load balancing (/tag/load-balancing/)*, *service discovery (/tag/service-discovery/)*, *ucp (/tag/ucp/)*, *universal control plane (/tag/universal-control-plane/)*

UNDERSTANDING DOCKER NETWORKING DRIVERS AND THEIR USE CASES

By Mark Church [https://blog.docker.com/author/mark-church/]

Technical Account Manager at Docker, Inc

---

10 Responses to "Understanding Docker Networking Drivers and their use cases"

Renato Isidii

December 21, 2016 (/2016/12/understanding-docker-networking-drivers-use-cases/#comment-378147)

Excellent post!!!

Reply (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/?replytocom=378147#respond)

Alan Dixon

December 22, 2016 (/2016/12/understanding-docker-networking-drivers-use-cases/#comment-378383)

Thanks for a great summary. Towards the end you mean important tenets instead of tenants.

Reply (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/?replytocom=378383#respond)

Mark Church

February 27, 2017 (/2016/12/understanding-docker-networking-drivers-use-cases/#comme

That is correct – tenets!

Reply (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-cases/?replytocom=385721#respond)

Tao Wang (http://blog.lab99.org/)

December 27, 2016 (/2016/12/understanding-docker-networking-drivers-use-cases/#comment-379038)

Hi, Mark, it's a great post, very informative.

However, there is a mistake at the end of the article:
http://success.docker.com/Datacenter/Apply/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Docke
(http://success.docker.com/Datacenter/Apply/Docker_Reference_Architecture%3A_Designing_Scalable%2C_Portable_Dock

At section "Tutorial App: MACVLAN Bridge Mode",

“`bash
#Creation of web container on host-A
host-A $ docker run -it –net macvlan –ip 192.168.0.4 -e 'DB=dog-db' -e 'ROLE=dog' –name dog-web chrch/web

#Creation of db container on host-B
host-B $ docker run -it –net macvlan –ip 192.168.0.5 –name dog-db redis
“`

`DB=dog-db` will not work, as the Service Discovery of MACVLAN only works `locally`, not `globally`. So, although both mac
`macvlan` on both hosts, the docker engine on host-A has no idea what is the IP of `dog-db` on host-B.

Reply (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/?replytocom=379

Mark Church

February 27, 2017 (/2016/12/understanding-docker-networking-drivers-use-cases/#comment-385722)

Thanks for the catch Tao. Service discovery with local scope drivers will only work locally with a single do
Swarm scope and global scope drivers distribute service discovery information across a cluster. We'll corr
version of the reference architecture.

Reply (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/?replytocom=385722#respond)

Jonathan

February 25, 2017 (/2016/12/understanding-docker-networking-drivers-use-cases/#comment-385468)

Hi,

Is it possible to create network policies with the overlay driver? Is this something that will soon be supported?

Reply (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/?replytocom=385468#respond)

Mark Church

February 27, 2017 (/2016/12/understanding-docker-networking-drivers-use-cases/#comment-385723)

Jonathan,

Currently overlay network policies can be defined via use of multiple overlay networks to provide granular segmentation. We are looking at providing better ways of defining network policy in future releases. Feel free to drop me an email for what kind of policy management would work for your use-cases -> church at docker.com

Reply (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/?replytocom=385723#respond)

Arthur

May 19, 2017 (/2016/12/understanding-docker-networking-drivers-use-cases/#comment-396935)

Hi,

Thanks for this article, it's a really good job ! However I have a question for you, do you know how many network interfaces is it possible to have for 1 container ?

Thanks,
Arthur

Reply (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/?replytocom=396935#respond)

rob

June 26, 2017 (/2016/12/understanding-docker-networking-drivers-use-cases/#comment-402419)

in the macvlan scenario, why can't the two servers have the same IP or be in same subnet ? from the vlan perspective , the router could be in two different vrfs upstream. I feel sometimes docker network people need to think outside of the box a bit. I have tried this configuration and docker requires a unique subnet per vlan. Why does docker even need to know the subnet in this scenario if it is just bridging between container and the external network ?

Reply (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/?replytocom=402419#respond)

Igor Podlesny (https://ru.etcinsider.com/)

January 2, 2018 (/2016/12/understanding-docker-networking-drivers-use-cases/#comment-420703)

Still don't see how all this (any of this) helps to accomplish very simple task: "attaching" a container to network as if it was an single IP node. With –net=host it won't be bound to single IP no way; with bridge you would have to tinker with port exposing and this is not even closely an equivalent of attaching node to network. All-in-all: an over-engineered something

P. S. There're also a few flaws in current Docker approach:
• it tries to set –gateway although it's not always valid and required even; gateway should be optional not mandatory
• you can't create /32 bridge network although it's pretty legitimate

Reply (https://blog.docker.com/2016/12/understanding-docker-networking-drivers-use-cases/?replytocom=420703#respond)

Leave a Reply