Network Configuration

TL;DR

When Docker starts, it creates a virtual interface named `docker0` on the host machine. It randomly chooses an address and subnet from the private range defined by RFC 1918 (http://tools.ietf.org/html/rfc1918) that are not in use on the host machine, and assigns it to `docker0`. Docker made the choice `172.17.42.1/16` when I started it a few minutes ago, for example — a 16-bit netmask providing 65,534 addresses for the host machine and its containers. The MAC address is generated using the IP address allocated to the container to avoid ARP collisions, using a range from `02:42:ac:11:00:00` to `02:42:ac:11:ff:ff`.

> ***Note:*** *This document discusses advanced networking configuration and options for Docker. In most cases you won't need this information. If you're looking to get started with a simpler explanation of Docker networking and an introduction to the concept of container linking see the Docker User Guide (/v1.5/userguide/dockerlinks/).*

But `docker0` is no ordinary interface. It is a virtual *Ethernet bridge* that automatically forwards packets between any other network interfaces that are attached to it. This lets containers communicate both with the host machine and with each other. Every time Docker creates a container, it creates a pair of “peer” interfaces that are like opposite ends of a pipe — a packet sent on one will be received on the other. It gives one of the peers to the container to become its `eth0` interface and keeps the other peer, with a unique name like `vethAQI2QT`, out in the namespace of the host machine. By binding every `veth*` interface to the `docker0` bridge, Docker creates a virtual subnet shared between the host machine and every Docker container.

The remaining sections of this document explain all of the ways that you can use Docker options and — in advanced cases — raw Linux networking commands to tweak, supplement, or entirely replace Docker's default networking configuration.

Quick Guide to the Options

Here is a quick list of the networking-related Docker command-line options, in case it helps you find the section below that you are looking for.

Some networking command-line options can only be supplied to the Docker server when it starts up, and cannot be changed once it is running:

- `-b BRIDGE` or `--bridge=BRIDGE` — see Building your own bridge
- `--bip=CIDR` — see Customizing docker0
- `--fixed-cidr` — see Customizing docker0
- `--fixed-cidr-v6` — see IPv6

- `-H SOCKET...` or `--host=SOCKET...` — This might sound like it would affect container networking, but it actually faces in the other direction: it tells the Docker server over what channels it should be willing to receive commands like “run container” and “stop container.”
- `--icc=true|false` — see Communication between containers
- `--ip=IP_ADDRESS` — see Binding container ports
- `--ipv6=true|false` — see IPv6
- `--ip-forward=true|false` — see Communication between containers and the wider world
- `--iptables=true|false` — see Communication between containers
- `--mtu=BYTES` — see Customizing docker0

There are two networking options that can be supplied either at startup or when `docker run` is invoked. When provided at startup, set the default value that `docker run` will later use if the options are not specified:

- `--dns=IP_ADDRESS...` — see Configuring DNS
- `--dns-search=DOMAIN...` — see Configuring DNS

Finally, several networking options can only be provided when calling `docker run` because they specify something specific to one container:

- `-h HOSTNAME` or `--hostname=HOSTNAME` — see Configuring DNS and How Docker networks a container
- `--link=CONTAINER_NAME_or_ID:ALIAS` — see Configuring DNS and Communication between containers
- `--net=bridge|none|container:NAME_or_ID|host` — see How Docker networks a container
- `--mac-address=MACADDRESS...` — see How Docker networks a container
- `-p SPEC` or `--publish=SPEC` — see Binding container ports
- `-P` or `--publish-all=true|false` — see Binding container ports

The following sections tackle all of the above topics in an order that moves roughly from simplest to most complex.

Configuring DNS

How can Docker supply each container with a hostname and DNS configuration, without having to build a custom image with the hostname written inside? Its trick is to overlay three crucial `/etc` files inside the container with virtual files where it can write fresh information. You can see this by running `mount` inside a container:

```
$$ mount
...
/dev/disk/by-uuid/1fec...ebdf on /etc/hostname type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/hosts type ext4 ...
/dev/disk/by-uuid/1fec...ebdf on /etc/resolv.conf type ext4 ...
...
```

This arrangement allows Docker to do clever things like keep `resolv.conf` up to date across all containers when the host machine receives new configuration over DHCP later. The exact details of how Docker maintains these files inside the container can change from one Docker version to the next, so you should leave the files themselves alone and use the following Docker options instead.

Four different options affect container domain name services.

- `-h HOSTNAME` or `--hostname=HOSTNAME` — sets the hostname by which the container knows itself. This is written into `/etc/hostname`, into `/etc/hosts` as the name of the container's host-facing IP address, and is the name that `/bin/bash` inside the container will display inside its prompt. But the hostname is not easy to see from outside the container. It will not appear in `docker ps` nor in the `/etc/hosts` file of any other container.

- `--link=CONTAINER_NAME_or_ID:ALIAS` — using this option as you `run` a container gives the new container's `/etc/hosts` an extra entry named `ALIAS` that points to the IP address of the container identified by `CONTAINER_NAME_or_ID`. This lets processes inside the new container connect to the hostname `ALIAS` without having to know its IP. The `--link=` option is discussed in more detail below, in the section Communication between containers. Because Docker may assign a different IP address to the linked containers on restart, Docker updates the `ALIAS` entry in the `/etc/hosts` file of the recipient containers.

- `--dns=IP_ADDRESS...` — sets the IP addresses added as `server` lines to the container's `/etc/resolv.conf` file. Processes in the container, when confronted with a hostname not in `/etc/hosts`, will connect to these IP addresses on port 53 looking for name resolution services.

- `--dns-search=DOMAIN...` — sets the domain names that are searched when a bare unqualified hostname is used inside of the container, by writing `search` lines into the container's `/etc/resolv.conf`. When a container process attempts to access `host` and the search domain `example.com` is set, for instance, the DNS logic will not only look up `host` but also `host.example.com`. Use `--dns-search=.` if you don't wish to set the search domain.

Note that Docker, in the absence of either of the last two options above, will make `/etc/resolv.conf` inside of each container look like the `/etc/resolv.conf` of the host machine where the `docker` daemon is running. You might wonder what happens when the host machine's `/etc/resolv.conf` file changes. The `docker` daemon has a file change notifier active which will watch for changes to the host DNS configuration. When the host file changes, all stopped containers which have a matching `resolv.conf` to the host will be updated immediately to this newest host configuration. Containers which are running when the host configuration changes will need to stop and start to pick up the host changes due to lack of a facility to ensure atomic writes of the `resolv.conf` file while the container is running. If the container's `resolv.conf` has been edited since it was started with the default configuration, no replacement will be attempted as it would overwrite the changes performed by the container. If the options (`--dns` or `--dns-search`) have been used to modify the default host configuration, then the replacement with an updated host's `/etc/resolv.conf` will not happen as well.

> ***Note**: For containers which were created prior to the implementation of the `/etc/resolv.conf` update feature in Docker 1.5.0: those containers will **not** receive updates when the host `resolv.conf` file changes. Only containers created with Docker 1.5.0 and above will utilize this auto-update feature.*

Communication between containers and the wider world

Whether a container can talk to the world is governed by two factors.

1. Is the host machine willing to forward IP packets? This is governed by the `ip_forward` system parameter. Packets can only pass between containers if this parameter is `1`. Usually you will simply leave the Docker server at its default setting `--ip-forward=true` and Docker will go set `ip_forward` to `1` for you when the server starts up. To check the setting or turn it on manually:

   ```
   $ cat /proc/sys/net/ipv4/ip_forward
   0
   $ echo 1 > /proc/sys/net/ipv4/ip_forward
   $ cat /proc/sys/net/ipv4/ip_forward
   1
   ```

   Many using Docker will want `ip_forward` to be on, to at least make communication *possible* between containers and the wider world.

   May also be needed for inter-container communication if you are in a multiple bridge setup.

2. Do your `iptables` allow this particular connection? Docker will never make changes to your system `iptables` rules if you set `--iptables=false` when the daemon starts. Otherwise the Docker server will append forwarding rules to the `DOCKER` filter chain.

Docker will not delete or modify any pre-existing rules from the `DOCKER` filter chain. This allows the user to create in advance any rules required to further restrict access to the containers.

Docker's forward rules permit all external source IPs by default. To allow only a specific IP or network to access the containers, insert a negated rule at the top of the `DOCKER` filter chain. For example, to restrict external access such that *only* source IP 8.8.8.8 can access the containers, the following rule could be added:

```
$ iptables -I DOCKER -i ext_if ! -s 8.8.8.8 -j DROP
```

Communication between containers

Whether two containers can communicate is governed, at the operating system level, by two factors.

1. Does the network topology even connect the containers' network interfaces? By default Docker will attach all containers to a single `docker0` bridge, providing a path for packets to travel between them. See the later sections of this document for other possible topologies.

2. Do your `iptables` allow this particular connection? Docker will never make changes to your system `iptables` rules if you set `--iptables=false` when the daemon starts. Otherwise the Docker server will add a default rule to the `FORWARD` chain with a blanket `ACCEPT` policy if you retain the default `--icc=true`, or else will set the policy to `DROP` if `--icc=false`.

It is a strategic question whether to leave `--icc=true` or change it to `--icc=false` (on Ubuntu, by editing the `DOCKER_OPTS` variable in `/etc/default/docker` and restarting the Docker server) so that `iptables` will protect other containers — and the main host — from having arbitrary ports probed or accessed by a container that gets compromised.

If you choose the most secure setting of `--icc=false`, then how can containers communicate in those cases where you *want* them to provide each other services?

The answer is the `--link=CONTAINER_NAME_or_ID:ALIAS` option, which was mentioned in the previous section because of its effect upon name services. If the Docker daemon is running with both `--icc=false` and `--iptables=true` then, when it sees `docker run` invoked with the `--link=` option, the Docker server will insert a pair of `iptables` `ACCEPT` rules so that the new container can connect to the ports exposed by the other container — the ports that it mentioned in the `EXPOSE` lines of its `Dockerfile`. Docker has more documentation on this subject — see the linking Docker containers (/v1.5/userguide/dockerlinks) page for further details.

> ***Note**: The value `CONTAINER_NAME` in `--link=` must either be an auto-assigned Docker name like `stupefied_pare` or else the name you assigned with `--name=` when you ran `docker run`. It cannot be a hostname, which Docker will not recognize in the context of the `--link=` option.*

You can run the `iptables` command on your Docker host to see whether the `FORWARD` chain has a default policy of `ACCEPT` or `DROP`:

```
# When --icc=false, you should see a DROP rule:

$ sudo iptables -L -n
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0
...

# When a --link= has been created under --icc=false,
# you should see port-specific ACCEPT rules overriding
# the subsequent DROP policy for all other packets:

$ sudo iptables -L -n
...
Chain FORWARD (policy ACCEPT)
target     prot opt source               destination
DOCKER     all  --  0.0.0.0/0            0.0.0.0/0
DROP       all  --  0.0.0.0/0            0.0.0.0/0

Chain DOCKER (1 references)
target     prot opt source               destination
ACCEPT     tcp  --  172.17.0.2           172.17.0.3           tcp spt:80
ACCEPT     tcp  --  172.17.0.3           172.17.0.2           tcp dpt:80
```

> ***Note**: Docker is careful that its host-wide `iptables` rules fully expose containers to each other's raw IP addresses, so connections from one container to another should always appear to be originating from the first container's own IP address.*

Binding container ports to the host

By default Docker containers can make connections to the outside world, but the outside world cannot connect to containers. Each outgoing connection will appear to originate from one of the host machine's own IP addresses thanks to an `iptables` masquerading rule on the host machine that the Docker server creates when it starts:

```
# You can see that the Docker server creates a
# masquerade rule that let containers connect
# to IP addresses in the outside world:

$ sudo iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16       !172.17.0.0/16
...
```

But if you want containers to accept incoming connections, you will need to provide special options when invoking `docker run`. These options are covered in more detail in the Docker User Guide (/v1.5/userguide/dockerlinks) page. There are two approaches.

First, you can supply `-P` or `--publish-all=true|false` to `docker run` which is a blanket operation that identifies every port with an `EXPOSE` line in the image's `Dockerfile` and maps it to a host port somewhere in the range 49153–65535. This tends to be a bit inconvenient, since you then have to run other `docker` sub-commands to learn which external port a given service was mapped to.

More convenient is the `-p SPEC` or `--publish=SPEC` option which lets you be explicit about exactly which external port on the Docker server — which can be any port at all, not just those in the 49153-65535 block — you want mapped to which port in the container.

Either way, you should be able to peek at what Docker has accomplished in your network stack by examining your NAT tables.

```
# What your NAT rules might look like when Docker
# is finished setting up a -P forward:

$ iptables -t nat -L -n
...
Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:49153 to:172.17.0.2:80

# What your NAT rules might look like when Docker
# is finished setting up a -p 80:80 forward:

Chain DOCKER (2 references)
target     prot opt source               destination
DNAT       tcp  --  0.0.0.0/0            0.0.0.0/0            tcp dpt:80 to:172.17.0.2:80
```

You can see that Docker has exposed these container ports on `0.0.0.0`, the wildcard IP address that will match any possible incoming port on the host machine. If you want to be more restrictive and only allow container services to be contacted through a specific external interface on the host machine, you have two choices. When you invoke `docker run` you can use either `-p IP:host_port:container_port` or `-p IP::port` to specify the external interface for one particular binding.

Or if you always want Docker port forwards to bind to one specific IP address, you can edit your system-wide Docker server settings (on Ubuntu, by editing `DOCKER_OPTS` in `/etc/default/docker`) and add the option `--ip=IP_ADDRESS`. Remember to restart your Docker server after editing this setting.

Again, this topic is covered without all of these low-level networking details in the Docker User Guide (/v1.5/userguide/dockerlinks/) document if you would like to use that as your port redirection reference instead.

IPv6

As we are running out of IPv4 addresses (http://en.wikipedia.org/wiki/IPv4_address_exhaustion) the IETF has standardized an IPv4 successor, Internet Protocol Version 6 (http://en.wikipedia.org/wiki/IPv6) , in RFC 2460 (https://www.ietf.org/rfc/rfc2460.txt). Both protocols, IPv4 and IPv6, reside on layer 3 of the OSI model (http://en.wikipedia.org/wiki/OSI_model).

IPv6 with Docker

By default, the Docker server configures the container network for IPv4 only. You can enable IPv4/IPv6 dualstack support by running the Docker daemon with the `--ipv6` flag. Docker will set up the bridge `docker0` with the IPv6 link-local address (http://en.wikipedia.org/wiki/Link-local_address) `fe80::1`.

By default, containers that are created will only get a link-local IPv6 address. To assign globally routable IPv6 addresses to your containers you have to specify an IPv6 subnet to pick the addresses from. Set the IPv6 subnet via the `--fixed-cidr-v6` parameter when starting Docker daemon:

```
docker -d --ipv6 --fixed-cidr-v6="2001:db8:1::/64"
```

The subnet for Docker containers should at least have a size of `/80`. This way an IPv6 address can end with the container's MAC address and you prevent NDP neighbor cache invalidation issues in the Docker layer.

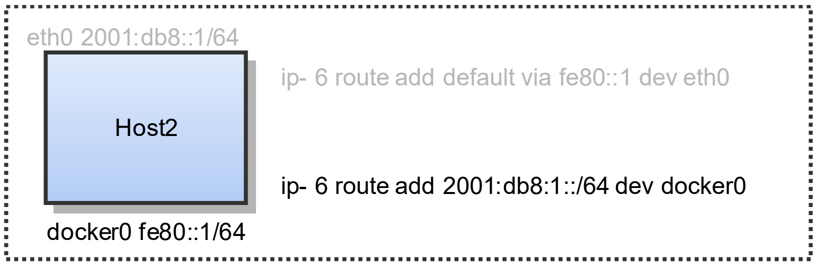
With the `--fixed-cidr-v6` parameter set Docker will add a new route to the routing table. Further IPv6 routing will be enabled (you may prevent this by starting Docker daemon with `--ip-forward=false`):

```
$ ip -6 route add 2001:db8:1::/64 dev docker0
$ sysctl net.ipv6.conf.default.forwarding=1
$ sysctl net.ipv6.conf.all.forwarding=1
```

All traffic to the subnet `2001:db8:1::/64` will now be routed via the `docker0` interface.

Be aware that IPv6 forwarding may interfere with your existing IPv6 configuration: If you are using Router Advertisements to get IPv6 settings for your host's interfaces you should set `accept_ra` to `2`. Otherwise IPv6 enabled forwarding will result in rejecting Router Advertisements. E.g., if you want to configure `eth0` via Router Advertisements you should set:

```
```
$ sysctl net.ipv6.conf.eth0.accept_ra=2
```
```


eth0 2001:db8::1/64
Host2
ip- 6 route add default via fe80::1 dev eth0
ip- 6 route add 2001:db8:1::/64 dev docker0
docker0 fe80::1/64

Every new container will get an IPv6 address from the defined subnet. Further a default route will be added via the gateway `fe80::1` on `eth0`:

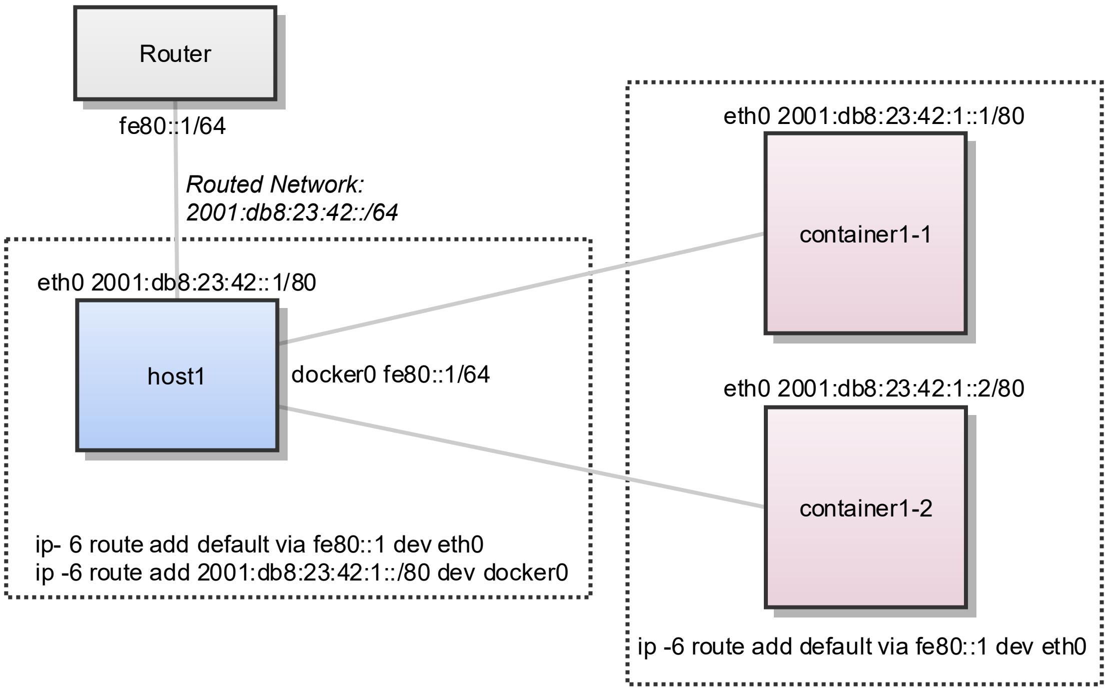
```
docker run -it ubuntu bash -c "ip -6 addr show dev eth0; ip -6 route show"

15: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500
   inet6 2001:db8:1:0:0:242:ac11:3/64 scope global
      valid_lft forever preferred_lft forever
   inet6 fe80::42:acff:fe11:3/64 scope link
      valid_lft forever preferred_lft forever

2001:db8:1::/64 dev eth0  proto kernel  metric 256
fe80::/64 dev eth0  proto kernel  metric 256
default via fe80::1 dev eth0  metric 1024
```

In this example the Docker container is assigned a link-local address with the network suffix `/64` (here: `fe80::42:acff:fe11:3/64`) and a globally routable IPv6 address (here: `2001:db8:1:0:0:242:ac11:3/64`). The container will create connections to addresses outside of the `2001:db8:1::/64` network via the link-local gateway at `fe80::1` on `eth0`.

Often servers or virtual machines get a `/64` IPv6 subnet assigned (e.g. `2001:db8:23:42::/64`). In this case you can split it up further and provide Docker a `/80` subnet while using a separate `/80` subnet for other applications on the host:

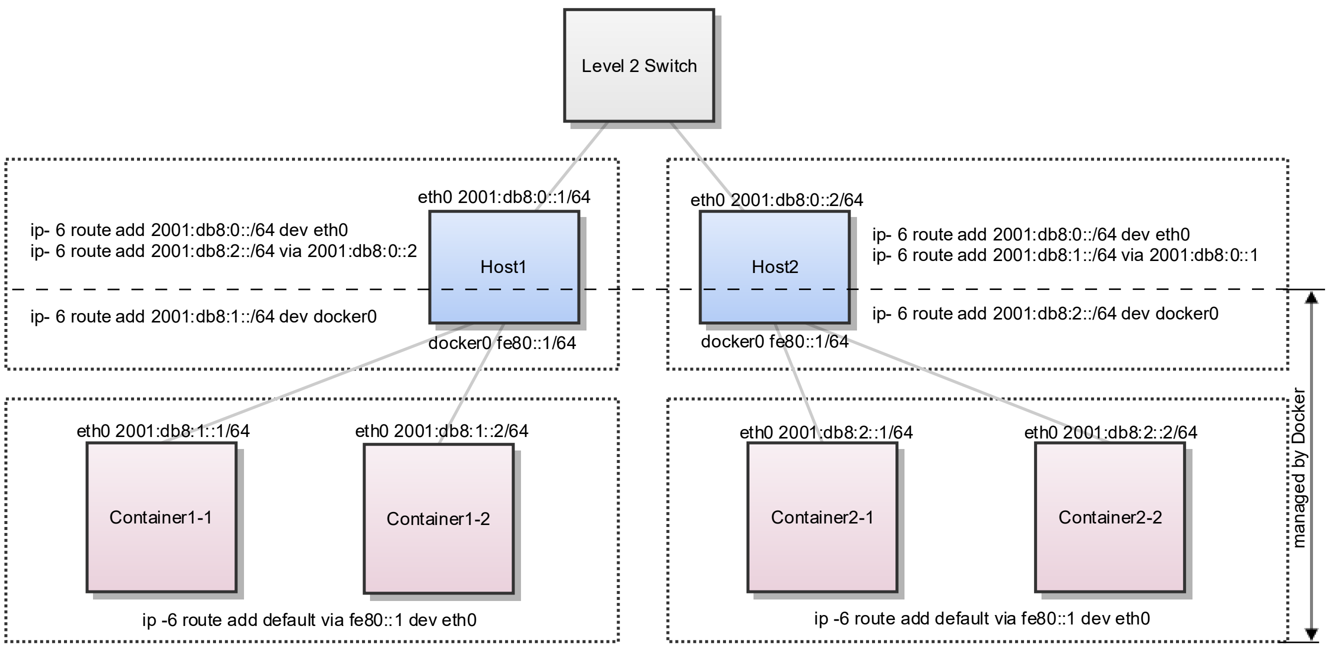

Router
fe80::1/64
Routed Network:
2001:db8:23:42::/64
eth0 2001:db8:23:42::1/80
host1
docker0 fe80::1/64
ip- 6 route add default via fe80::1 dev eth0
ip -6 route add 2001:db8:23:42:1::/80 dev docker0
eth0 2001:db8:23:42:1::1/80
container1-1
eth0 2001:db8:23:42:1::2/80
container1-2
ip -6 route add default via fe80::1 dev eth0

In this setup the subnet `2001:db8:23:42::/80` with a range from `2001:db8:23:42:0:0:0:0` to `2001:db8:23:42:0:ffff:ffff:ffff` is attached to `eth0`, with the host listening at `2001:db8:23:42::1`. The subnet `2001:db8:23:42:1::/80` with an address range from `2001:db8:23:42:1:0:0:0` to `2001:db8:23:42:1:ffff:ffff:ffff` is attached to `docker0` and will be used by containers.

Docker IPv6 Cluster

Switched Network Environment

Using routable IPv6 addresses allows you to realize communication between containers on different hosts. Let's have a look at a simple Docker IPv6 cluster example:


Level 2 Switch
eth0 2001:db8:0::1/64
ip- 6 route add 2001:db8:0::/64 dev eth0
ip- 6 route add 2001:db8:2::/64 via 2001:db8:0::2
Host1
ip- 6 route add 2001:db8:1::/64 dev docker0
docker0 fe80::1/64
eth0 2001:db8:0::2/64
Host2
ip- 6 route add 2001:db8:0::/64 dev eth0
ip- 6 route add 2001:db8:1::/64 via 2001:db8:0::1
ip- 6 route add 2001:db8:2::/64 dev docker0
docker0 fe80::1/64
eth0 2001:db8:1::1/64
Container1-1
eth0 2001:db8:1::2/64
Container1-2
ip -6 route add default via fe80::1 dev eth0
eth0 2001:db8:2::1/64
Container2-1
eth0 2001:db8:2::2/64
Container2-2
ip -6 route add default via fe80::1 dev eth0
managed by Docker
containers' link-local addresses are not displayed

The Docker hosts are in the `2001:db8:0::/64` subnet. Host1 is configured to provide addresses from the `2001:db8:1::/64` subnet to its containers. It has three routes configured:

- Route all traffic to `2001:db8:0::/64` via `eth0`
- Route all traffic to `2001:db8:1::/64` via `docker0`
- Route all traffic to `2001:db8:2::/64` via Host2 with IP `2001:db8::2`

Host1 also acts as a router on OSI layer 3. When one of the network clients tries to contact a target that is specified in Host1's routing table Host1 will forward the traffic accordingly. It acts as a router for all networks it knows: `2001:db8::/64`, `2001:db8:1::/64` and `2001:db8:2::/64`.

On Host2 we have nearly the same configuration. Host2's containers will get IPv6 addresses from `2001:db8:2::/64`. Host2 has three routes configured:

- Route all traffic to `2001:db8:0::/64` via `eth0`
- Route all traffic to `2001:db8:2::/64` via `docker0`
- Route all traffic to `2001:db8:1::/64` via Host1 with IP `2001:db8:0::1`

The difference to Host1 is that the network `2001:db8:2::/64` is directly attached to the host via its `docker0` interface whereas it reaches `2001:db8:1::/64` via Host1's IPv6 address `2001:db8::1`.

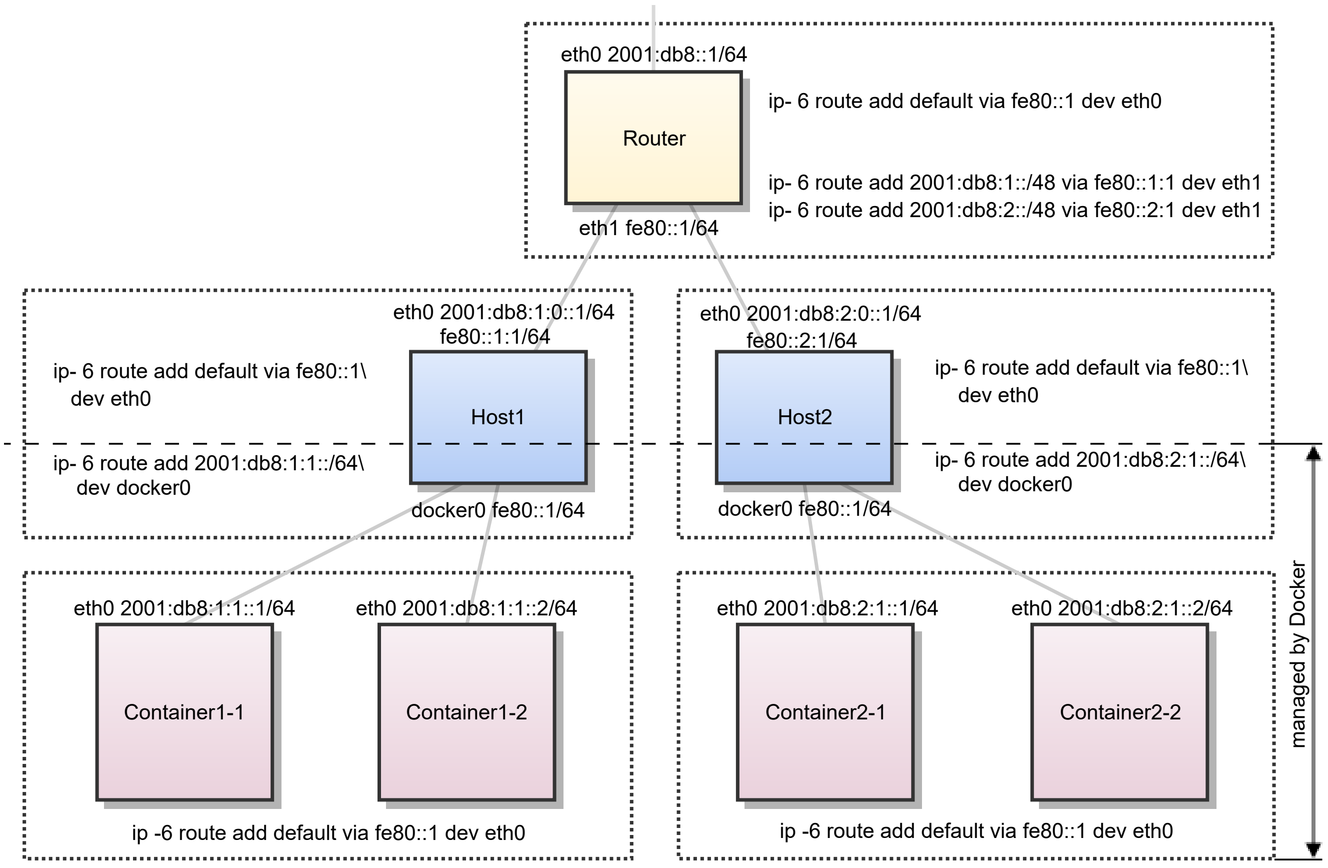
This way every container is able to contact every other container. The containers `Container1-*` share the same subnet and contact each other directly. The traffic between `Container1-*` and `Container2-*` will be routed via Host1 and Host2 because those containers do not share the same subnet.

In a switched environment every host has to know all routes to every subnet. You always have to update the hosts' routing tables once you add or remove a host to the cluster.

Every configuration in the diagram that is shown below the dashed line is handled by Docker: The `docker0` bridge IP address configuration, the route to the Docker subnet on the host, the container IP addresses and the routes on the containers. The configuration above the line is up to the user and can be adapted to the individual environment.

Routed Network Environment

In a routed network environment you replace the level 2 switch with a level 3 router. Now the hosts just have to know their default gateway (the router) and the route to their own containers (managed by Docker). The router holds all routing information about the Docker subnets. When you add or remove a host to this environment you just have to update the routing table in the router - not on every host.


eth0 2001:db8::1/64
Router
ip- 6 route add default via fe80::1 dev eth0
ip- 6 route add 2001:db8:1::/48 via fe80::1:1 dev eth1
ip- 6 route add 2001:db8:2::/48 via fe80::2:1 dev eth1
eth1 fe80::1/64
eth0 2001:db8:1:0::1/64
fe80::1:1/64
ip- 6 route add default via fe80::1\
dev eth0
Host1
ip- 6 route add 2001:db8:1:1::/64\
dev docker0
docker0 fe80::1/64
eth0 2001:db8:2:0::1/64
fe80::2:1/64
ip- 6 route add default via fe80::1\
dev eth0
Host2
ip- 6 route add 2001:db8:2:1::/64\
dev docker0
docker0 fe80::1/64
eth0 2001:db8:1:1::1/64
Container1-1
eth0 2001:db8:1:1::2/64
Container1-2
ip -6 route add default via fe80::1 dev eth0
eth0 2001:db8:2:1::1/64
Container2-1
eth0 2001:db8:2:1::2/64
Container2-2
ip -6 route add default via fe80::1 dev eth0
managed by Docker

containers' link-local addresses are not displayed

In this scenario containers of the same host can communicate directly with each other. The traffic between containers on different hosts will be routed via their hosts and the router. For example packet from `Container1-1` to `Container2-1` will be routed through `Host1`, `Router` and `Host2` until it arrives at `Container2-1`.

To keep the IPv6 addresses short in this example a `/48` network is assigned to every host. The hosts use a `/64` subnet of this for its own services and one for Docker. When adding a third host you would add a route for the subnet `2001:db8:3::/48` in the router and configure Docker on Host3 with `--fixed-cidr-v6=2001:db8:3:1::/64`.

Remember the subnet for Docker containers should at least have a size of `/80`. This way an IPv6 address can end with the container's MAC address and you prevent NDP neighbor cache invalidation issues in the Docker layer. So if you have a `/64` for your whole environment use `/68` subnets for the hosts and `/80` for the containers. This way you can use 4096 hosts with 16 `/80` subnets each.

Every configuration in the diagram that is visualized below the dashed line is handled by Docker: The `docker0` bridge IP address configuration, the route to the Docker subnet on the host, the container IP addresses and the routes on the containers. The configuration above the line is up to the user and can be adapted to the individual environment.

Customizing docker0

By default, the Docker server creates and configures the host system's `docker0` interface as an *Ethernet bridge* inside the Linux kernel that can pass packets back and forth between other physical or virtual network interfaces so that they behave as a single Ethernet network.

Docker configures `docker0` with an IP address, netmask and IP allocation range. The host machine can both receive and send packets to containers connected to the bridge, and gives it an MTU — the *maximum transmission unit* or largest packet length that the interface will allow — of either 1,500 bytes or else a more specific value copied from the Docker host's interface that supports its default route. These options are configurable at server startup:

- `--bip=CIDR` — supply a specific IP address and netmask for the `docker0` bridge, using standard CIDR notation like `192.168.1.5/24`.

- `--fixed-cidr=CIDR` — restrict the IP range from the `docker0` subnet, using the standard CIDR notation like `172.167.1.0/28`. This range must be and IPv4 range for fixed IPs (ex: 10.20.0.0/16) and must be a subset of the bridge IP range (`docker0` or set using `--bridge`). For example with `--fixed-cidr=192.168.1.0/25`, IPs for your containers will be chosen from the first half of `192.168.1.0/24` subnet.

- `--mtu=BYTES` — override the maximum packet length on `docker0`.

On Ubuntu you would add these to the `DOCKER_OPTS` setting in `/etc/default/docker` on your Docker host and restarting the Docker service.

Once you have one or more containers up and running, you can confirm that Docker has properly connected them to the `docker0` bridge by running the `brctl` command on the host machine and looking at the `interfaces` column of the output. Here is a host with two different containers connected:

```
# Display bridge info

$ sudo brctl show
bridge name     bridge id               STP enabled     interfaces
docker0         8000.3a1d7362b4ee       no              veth65f9
                                                        vethdda6
```

If the `brctl` command is not installed on your Docker host, then on Ubuntu you should be able to run `sudo apt-get install bridge-utils` to install it.

Finally, the `docker0` Ethernet bridge settings are used every time you create a new container. Docker selects a free IP address from the range available on the bridge each time you `docker run` a new container, and configures the container's `eth0` interface with that IP address and the bridge's netmask. The Docker host's own IP address on the bridge is used as the default gateway by which each container reaches the rest of the Internet.

```
# The network, as seen from a container

$ sudo docker run -i -t --rm base /bin/bash

$$ ip addr show eth0
24: eth0: <BROADCAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state UP group default qlen 1000
    link/ether 32:6f:e0:35:57:91 brd ff:ff:ff:ff:ff:ff
    inet 172.17.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::306f:e0ff:fe35:5791/64 scope link
       valid_lft forever preferred_lft forever

$$ ip route
default via 172.17.42.1 dev eth0
172.17.0.0/16 dev eth0  proto kernel  scope link  src 172.17.0.3

$$ exit
```

Remember that the Docker host will not be willing to forward container packets out on to the Internet unless its `ip_forward` system setting is `1` — see the section above on Communication between containers for details.

Building your own bridge

If you want to take Docker out of the business of creating its own Ethernet bridge entirely, you can set up your own bridge before starting Docker and use `-b BRIDGE` or `--bridge=BRIDGE` to tell Docker to use your bridge instead. If you already have Docker up and running with its old `docker0` still configured, you will probably want to begin by stopping the service and removing the interface:

```
# Stopping Docker and removing docker0

$ sudo service docker stop
$ sudo ip link set dev docker0 down
$ sudo brctl delbr docker0
$ sudo iptables -t nat -F POSTROUTING
```

Then, before starting the Docker service, create your own bridge and give it whatever configuration you want. Here we will create a simple enough bridge that we really could just have used the options in the previous section to customize `docker0`, but it will be enough to illustrate the technique.

```
# Create our own bridge

$ sudo brctl addbr bridge0
$ sudo ip addr add 192.168.5.1/24 dev bridge0
$ sudo ip link set dev bridge0 up

# Confirming that our bridge is up and running

$ ip addr show bridge0
4: bridge0: <BROADCAST,MULTICAST> mtu 1500 qdisc noop state UP group default
    link/ether 66:38:d0:0d:76:18 brd ff:ff:ff:ff:ff:ff
    inet 192.168.5.1/24 scope global bridge0
       valid_lft forever preferred_lft forever

# Tell Docker about it and restart (on Ubuntu)

$ echo 'DOCKER_OPTS="-b=bridge0"' >> /etc/default/docker
$ sudo service docker start

# Confirming new outgoing NAT masquerade is set up

$ sudo iptables -t nat -L -n
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  192.168.5.0/24      0.0.0.0/0
```

The result should be that the Docker server starts successfully and is now prepared to bind containers to the new bridge. After pausing to verify the bridge's configuration, try creating a container — you will see that its IP address is in your new IP address range, which Docker will have auto-detected.

Just as we learned in the previous section, you can use the `brctl show` command to see Docker add and remove interfaces from the bridge as you start and stop containers, and can run `ip addr` and `ip route` inside a container to see that it has been given an address in the bridge's IP address range and has been told to use the Docker host's IP address on the bridge as its default gateway to the rest of the Internet.

How Docker networks a container

While Docker is under active development and continues to tweak and improve its network configuration logic, the shell commands in this section are rough equivalents to the steps that Docker takes when configuring networking for each new container.

Let's review a few basics.

To communicate using the Internet Protocol (IP), a machine needs access to at least one network interface at which packets can be sent and received, and a routing table that defines the range of IP addresses reachable through that interface. Network interfaces do not have to be physical devices. In fact, the `lo` loopback interface available on every Linux machine (and inside each Docker container) is entirely virtual — the Linux kernel simply copies loopback packets directly from the sender's memory into the receiver's memory.

Docker uses special virtual interfaces to let containers communicate with the host machine — pairs of virtual interfaces called “peers” that are linked inside of the host machine's kernel so that packets can travel between them. They are simple to create, as we will see in a moment.

The steps with which Docker configures a container are:

1. Create a pair of peer virtual interfaces.

2. Give one of them a unique name like `veth65f9`, keep it inside of the main Docker host, and bind it to `docker0` or whatever bridge Docker is supposed to be using.

3. Toss the other interface over the wall into the new container (which will already have been provided with an `lo` interface) and rename it to the much prettier name `eth0` since, inside of the container's separate and unique network interface namespace, there are no physical interfaces with which this name could collide.

4. Set the interface's MAC address according to the `--mac-address` parameter or generate a random one.

5. Give the container's `eth0` a new IP address from within the bridge's range of network addresses, and set its default route to the IP address that the Docker host owns on the bridge. If available the IP address is generated from the MAC address. This prevents ARP cache invalidation problems, when a new container comes up with an IP used in the past by another container with another MAC.

With these steps complete, the container now possesses an `eth0` (virtual) network card and will find itself able to communicate with other containers and the rest of the Internet.

You can opt out of the above process for a particular container by giving the `--net=` option to `docker run`, which takes four possible values.

- `--net=bridge` — The default action, that connects the container to the Docker bridge as described above.

- `--net=host` — Tells Docker to skip placing the container inside of a separate network stack. In essence, this choice tells Docker to **not containerize the container's networking**! While container processes will still be confined to their own filesystem and process list and resource limits, a quick `ip addr` command will show you that, network-wise, they live “outside” in the main Docker host and have full access to its network interfaces. Note that this does **not** let the container reconfigure the host network stack — that would require `--privileged=true` — but it does let container processes open low-numbered ports like any other root process. It also allows the container to access local network services like D-bus. This can lead to processes in the container being able to do unexpected things like restart your computer (https://github.com/docker/docker/issues/6401). You should use this option with caution.

- `--net=container:NAME_or_ID` — Tells Docker to put this container's processes inside of the network stack that has already been created inside of another container. The new container's processes will be confined to their own filesystem and process list and resource limits, but will share the same IP address and port numbers as the first container, and processes on the two containers will be able to connect to each other over the loopback interface.

- `--net=none` — Tells Docker to put the container inside of its own network stack but not to take any steps to configure its network, leaving you free to build any of the custom configurations explored in the last few sections of this document.

To get an idea of the steps that are necessary if you use `--net=none` as described in that last bullet point, here are the commands that you would run to reach roughly the same configuration as if you had let Docker do all of the configuration:

```
# At one shell, start a container and
# leave its shell idle and running

$ sudo docker run -i -t --rm --net=none base /bin/bash
root@63f36fc01b5f:/#

# At another shell, learn the container process ID
# and create its namespace entry in /var/run/netns/
# for the "ip netns" command we will be using below

$ sudo docker inspect -f '{{.State.Pid}}' 63f36fc01b5f
2778
$ pid=2778
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/$pid/ns/net /var/run/netns/$pid

# Check the bridge's IP address and netmask

$ ip addr show docker0
21: docker0: ...
inet 172.17.42.1/16 scope global docker0
...

# Create a pair of "peer" interfaces A and B,
# bind the A end to the bridge, and bring it up

$ sudo ip link add A type veth peer name B
$ sudo brctl addif docker0 A
$ sudo ip link set A up

# Place B inside the container's network namespace,
# rename to eth0, and activate it with a free IP

$ sudo ip link set B netns $pid
$ sudo ip netns exec $pid ip link set dev B name eth0
$ sudo ip netns exec $pid ip link set eth0 address 12:34:56:78:9a:bc
$ sudo ip netns exec $pid ip link set eth0 up
$ sudo ip netns exec $pid ip addr add 172.17.42.99/16 dev eth0
$ sudo ip netns exec $pid ip route add default via 172.17.42.1
```

At this point your container should be able to perform networking operations as usual.

When you finally exit the shell and Docker cleans up the container, the network namespace is destroyed along with our virtual `eth0` — whose destruction in turn destroys interface `A` out in the Docker host and automatically un-registers it from the `docker0` bridge. So everything gets cleaned up without our having to run any extra commands! Well, almost everything:

```
# Clean up dangling symlinks in /var/run/netns

find -L /var/run/netns -type l -delete
```

Also note that while the script above used modern `ip` command instead of old deprecated wrappers like `ipconfig` and `route`, these older commands would also have worked inside of our container. The `ip addr` command can be typed as `ip a` if you are in a hurry.

Finally, note the importance of the `ip netns exec` command, which let us reach inside and configure a network namespace as root. The same commands would not have worked if run inside of the container, because part of safe containerization is that Docker strips container processes of the right to configure their own networks. Using `ip netns exec` is what let us finish up the configuration without having to take the dangerous step of running the container itself with `--privileged=true`.

Tools and Examples

Before diving into the following sections on custom network topologies, you might be interested in glancing at a few external tools or examples of the same kinds of configuration. Here are two:

- Jérôme Petazzoni has created a `pipework` shell script to help you connect together containers in arbitrarily complex scenarios: https://github.com/jpetazzo/pipework (https://github.com/jpetazzo/pipework)

- Brandon Rhodes has created a whole network topology of Docker containers for the next edition of Foundations of Python Network Programming that includes routing, NAT'd firewalls, and servers that offer HTTP, SMTP, POP, IMAP, Telnet, SSH, and FTP: https://github.com/brandon-rhodes/fopnp/tree/m/playground (https://github.com/brandon-rhodes/fopnp/tree/m/playground)

Both tools use networking commands very much like the ones you saw in the previous section, and will see in the following sections.

Building a point-to-point connection

By default, Docker attaches all containers to the virtual subnet implemented by `docker0`. You can create containers that are each connected to some different virtual subnet by creating your own bridge as shown in Building your own bridge, starting each container with `docker run --net=none`, and then attaching the containers to your bridge with the shell commands shown in How Docker networks a container.

But sometimes you want two particular containers to be able to communicate directly without the added complexity of both being bound to a host-wide Ethernet bridge.

The solution is simple: when you create your pair of peer interfaces, simply throw *both* of them into containers, and configure them as classic point-to-point links. The two containers will then be able to communicate directly (provided you manage to tell each container the other's IP address, of course). You might adjust the instructions of the previous section to go something like this:

```
# Start up two containers in two terminal windows

$ sudo docker run -i -t --rm --net=none base /bin/bash
root@1f1f4c1f931a:/#

$ sudo docker run -i -t --rm --net=none base /bin/bash
root@12e343489d2f:/#

# Learn the container process IDs
# and create their namespace entries

$ sudo docker inspect -f '{{.State.Pid}}' 1f1f4c1f931a
2989
$ sudo docker inspect -f '{{.State.Pid}}' 12e343489d2f
3004
$ sudo mkdir -p /var/run/netns
$ sudo ln -s /proc/2989/ns/net /var/run/netns/2989
$ sudo ln -s /proc/3004/ns/net /var/run/netns/3004

# Create the "peer" interfaces and hand them out

$ sudo ip link add A type veth peer name B

$ sudo ip link set A netns 2989
$ sudo ip netns exec 2989 ip addr add 10.1.1.1/32 dev A
$ sudo ip netns exec 2989 ip link set A up
$ sudo ip netns exec 2989 ip route add 10.1.1.2/32 dev A

$ sudo ip link set B netns 3004
$ sudo ip netns exec 3004 ip addr add 10.1.1.2/32 dev B
$ sudo ip netns exec 3004 ip link set B up
$ sudo ip netns exec 3004 ip route add 10.1.1.1/32 dev B
```

The two containers should now be able to ping each other and make connections successfully. Point-to-point links like this do not depend on a subnet nor a netmask, but on the bare assertion made by `ip route` that some other single IP address is connected to a particular network interface.

Note that point-to-point links can be safely combined with other kinds of network connectivity — there is no need to start the containers with `--net=none` if you want point-to-point links to be an addition to the container's normal networking instead of a replacement.

A final permutation of this pattern is to create the point-to-point link between the Docker host and one container, which would allow the host to communicate with that one container on some single IP address and thus communicate "out-of-band" of the bridge that connects the other, more usual containers. But unless you have very specific networking needs that drive you to such a solution, it is probably far preferable to use `--icc=false` to lock down inter-container communication, as we explored earlier.

Editing networking config files

Starting with Docker v.1.2.0, you can now edit `/etc/hosts`, `/etc/hostname` and `/etc/resolve.conf` in a running container. This is useful if you need to install bind or other services that might override one of those files.

Note, however, that changes to these files will not be saved by `docker commit`, nor will they be saved during `docker run`. That means they won't be saved in the image, nor will they persist when a container is restarted; they will only "stick" in a running container.