O'REILLY®

Latest Articles / Infrastructure

What is Docker networking?

Learn the basics of setting up Docker single-host networking.

By Michael Hausenblas

April 11, 2016

Train connection (source: Wikimedia Commons)

> For more on Docker networking, including an overview of multi-host networking, see the free ebook Docker Networking and Service Discovery, by Michael Hausenblas.

When you start working with Docker at scale, you all of a sudden need to know a lot about networking. As an introduction to networking with Docker, were going to start small, and show how quickly you need to start thinking about how to manage connections between containers. A Docker container needs a host to run on. This can either be a physical machine (e.g., a bare-metal server in your on-premise datacenter) or a VM either on-prem or in the cloud. The host has the Docker daemon and client running, as depicted in Figure 1, which enables you to interact with a Docker registry on the one hand (to pull/push Docker images), and on the other hand, allows you to start, stop, and inspect containers.

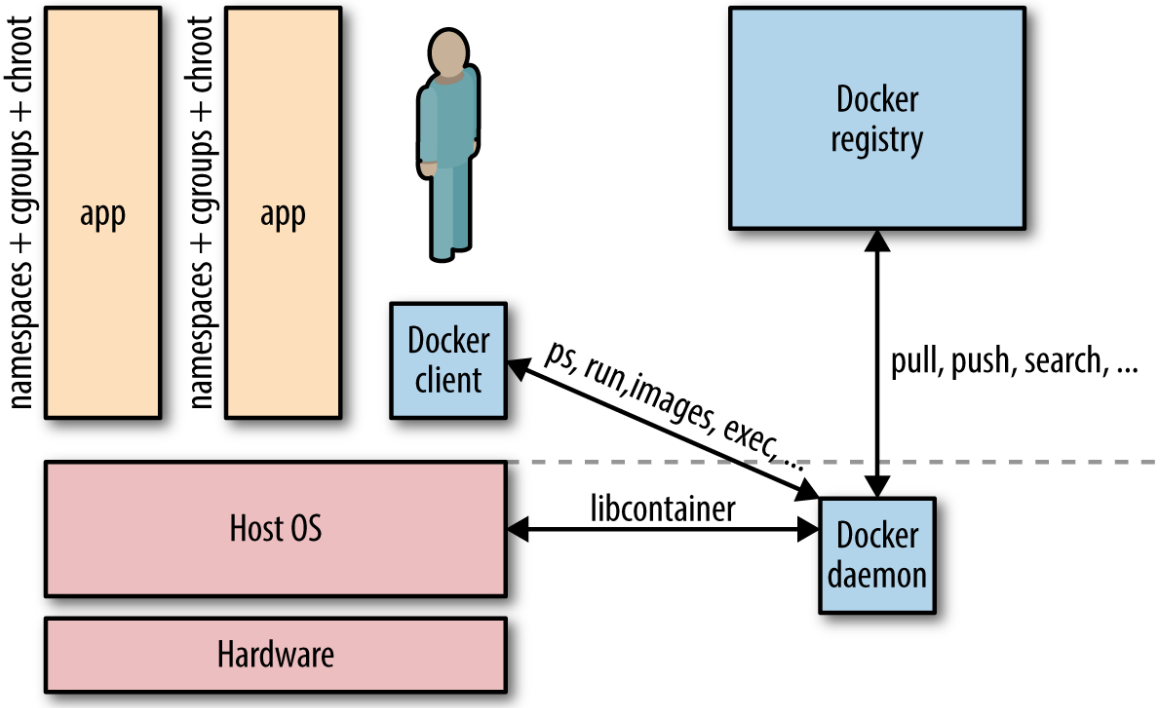

Figure 1. Simplified Docker architecture (single host)

The relationship between a host and containers is `1:N`. This means that one host typically has several containers running on it. For example, Facebook reports thatdepending on how beefy the machine isit sees on average some 10 to 40 containers per host running. And heres another data point: at Mesosphere, we found in various load tests on bare metal that not more than around 250 containers per host would be possible.[1]

No matter if you have a single-host deployment or use a cluster of machines, you will almost always have to deal with networking:

- For most *single-host deployments*, the question boils down to data exchange via a shared volume versus data exchange through networking (HTTP-based or otherwise). Although a Docker data volume is simple to use, it also introduces tight coupling, meaning that it will be harder to turn a single-host deployment into a multihost deployment. Naturally, the upside of shared volumes is speed.
- In *multihost deployments*, you need to consider two aspects: how are containers communicating within a host and how does the communication paths look between different hosts. Both performance considerations and security aspects will likely influence your design decisions. Multihost deployments usually become necessary either when the capacity of a single host is insufficient (see the earlier discussion on average and maximal number of containers on a host) or when one wants to employ distributed systems such as Apache Spark, HDFS, or Cassandra.

Distributed Systems and Data Locality

The basic idea behind using a distributed system (for computation or storage) is to benefit from parallel processing, usually together with data locality. By data locality I mean the principle to ship the code to where the data is rather than the (traditional) other way around. Think about the following for a moment: if your dataset size is in the TB and your code size is in the MB, its more efficient to move the code across the cluster than transferring TBs of data to a central processing place. In addition to being able to process things in parallel, you usually gain fault tolerance with distributed systems, as parts of the system can continue to work more or less independently.

Simply put, Docker networking is the native container SDN solution you have at your disposal when working with Docker. In a nutshell, there are four modes available for Docker networking: bridge mode, host mode, container mode, or no networking.[2] We will have a closer look at each of those modes relevant for a single-host setup and conclude at the end of this article with some general topics such as security.

Bridge Mode Networking

In this mode (see Figure 2), the Docker daemon creates `docker0`, a virtual Ethernet bridge that automatically forwards packets between any other network interfaces that are attached to it. By default, the daemon then connects all containers on a host to this internal network through creating a pair of peer interfaces, assigning one of the peers to become the containers

`eth0` interface and other peer in the namespace of the host, as well as assigning an IP address/subnet from the private IP range to the bridge (Example 1).

Example 1. Docker bridge mode networking in action

```
$ docker run -d -P --net=bridge nginx:1.9.1
$ docker ps
CONTAINER ID   IMAGE                  COMMAND    CREATED
STATUS         PORTS                  NAMES
17d447b7425d   nginx:1.9.1            nginx -g   19 seconds ago
Up 18 seconds  0.0.0.0:49153->443/tcp,
               0.0.0.0:49154->80/tcp  trusting_feynman
```

Note

Because bridge mode is the Docker default, you could have equally used `docker run -d -P nginx:1.9.1` in Example 1. If you do not use `-P` (which publishes all exposed ports of the container) or `-p host_port:container_port` (which publishes a specific port), the IP packets will not be routable to the container outside of the host.

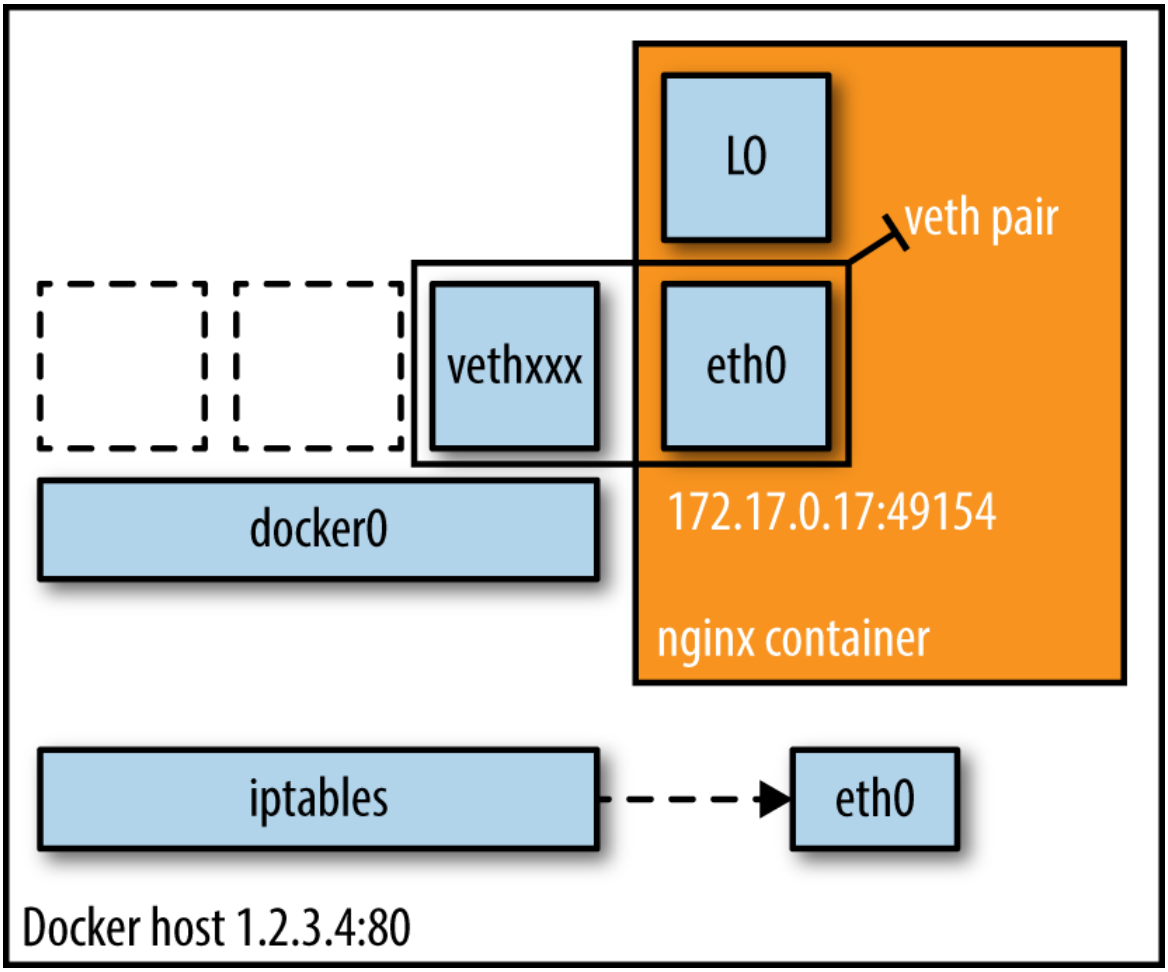

Figure 2. Bridge mode networking setup

Host Mode Networking

This mode effectively disables network isolation of a Docker container. Because the container shares the networking namespace of the host, it is directly exposed to the public network; consequently, you need to carry out the coordination via port mapping.

Example 2. Docker host mode networking in action

```
$ docker run -d --net=host ubuntu:14.04 tail -f /dev/null
$ ip addr | grep -A 2 eth0:
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP group de
    link/ether 06:58:2b:07:d5:f3 brd ff:ff:ff:ff:ff:ff
    inet **10.0.7.197**/22 brd 10.0.7.255 scope global dynamic eth0

$ docker ps
CONTAINER ID  IMAGE         COMMAND  CREATED
STATUS        PORTS         NAMES
b44d7d5d3903  ubuntu:14.04  tail -f  2 seconds ago
Up 2 seconds                jovial_blackwell
```

```
$ docker exec -it b44d7d5d3903 ip addr
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc mq state UP group de
    link/ether 06:58:2b:07:d5:f3 brd ff:ff:ff:ff:ff:ff
    inet **10.0.7.197**/22 brd 10.0.7.255 scope global dynamic eth0
```

And there we have it: as shown in Example 2, the container has the same IP address as the host, namely `10.0.7.197`.

In Figure 3, we see that when using host mode networking, the container effectively inherits the IP address from its host. This mode is faster than the bridge mode (because there is no routing overhead), but it exposes the container directly to the public network, with all its security implications.

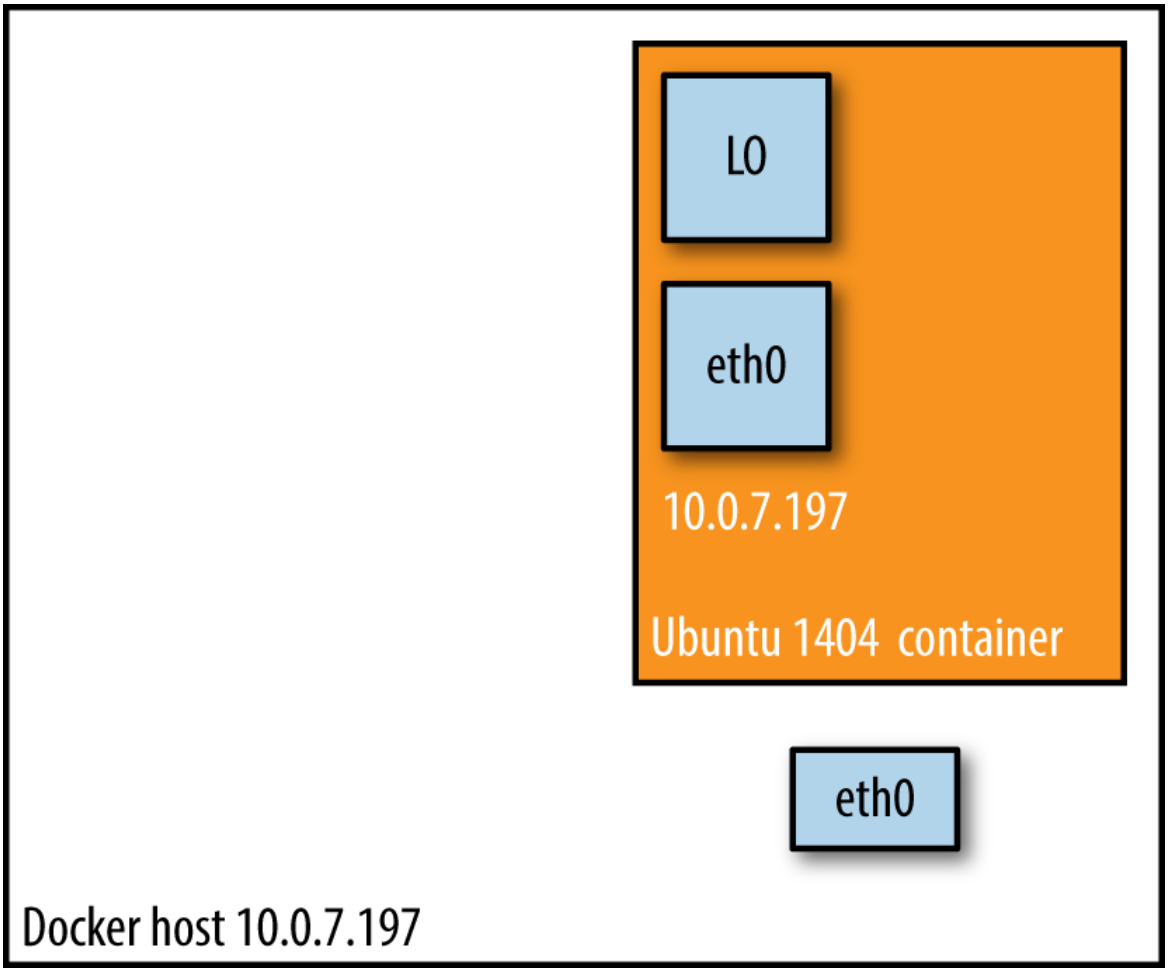

Figure 3. Docker host mode networking setup

Container Mode Networking

In this mode, you tell Docker to reuse the networking namespace of another container. In general, this mode is useful when you want to provide custom network stacks. Indeed, this mode is also what Kubernetes networking leverages.

Example 3. Docker container mode networking in action

```
$ docker run -d -P --net=bridge nginx:1.9.1
$ docker ps
CONTAINER ID  IMAGE        COMMAND   CREATED         STATUS
PORTS                      NAMES
eb19088be8a0  nginx:1.9.1  nginx -g  3 minutes ago   Up 3 minutes
0.0.0.0:32769->80/tcp,
0.0.0.0:32768->443/tcp     admiring_engelbart
$ docker exec -it admiring_engelbart ip addr
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet **172.17.0.3**/16 scope global eth0

$ docker run -it --net=container:admiring_engelbart ubuntu:14.04 ip addr
...
8: eth0@if9: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001 qdisc noqueue state UP
    link/ether 02:42:ac:11:00:03 brd ff:ff:ff:ff:ff:ff
    inet **172.17.0.3**/16 scope global eth0
```

The result (as shown in Example 3) is what we would have expected: the second container, started with `--net=container`, has the same IP address as the first container with the glorious auto-assigned name `admiring_engelbart`, namely `172.17.0.3`.

No Networking

This mode puts the container inside of its own network stack but doesnt configure it. Effectively, this turns off networking and is useful for two cases: either for containers that dont need a network (such as batch jobs writing to a disk volume) or if you want to set up your custom networking.

Example 4. Docker no-networking in action

```
$ docker run -d -P --net=none nginx:1.9.1
$ docker ps
CONTAINER ID  IMAGE          COMMAND   CREATED
STATUS        PORTS          NAMES
d8c26d68037c  nginx:1.9.1    nginx -g  2 minutes ago
Up 2 minutes                 grave_perlman
```

```
$  docker inspect d8c26d68037c | grep IPAddress
    "IPAddress": "",
    "SecondaryIPAddresses": null,
```

And as you can see in Example 4, there is no network configuredprecisely as we would have hoped for.

You can read more about networking and learn about configuration options on the excellent Docker docs pages.

> **Note**
>
> All Docker commands in this article have been executed in a CoreOS environment with both Docker client and server on version 1.7.1.

Wrapping It Up

Beyond the four basic Docker single-host networking modes discussed above, there are a few other aspects you should be aware of (and which are equally relevant for multihost deployments). These are covered below.

Allocating IP addresses
Manually allocating IP addresses when containers come and go frequently and in large numbers is not sustainable.[3] The bridge mode takes care of this issue to a certain extent. To prevent ARP collisions on a local network, the Docker daemon generates a random MAC address from the allocated IP address.

Allocating ports
You will find yourself either in the fixed-port-allocation or in the dynamically-port-allocation camp. This can be per service/application or as a global strategy, but you must make up your mind. Remember that, for bridge mode, Docker can automatically assign (UDP or TCP) ports and consequently make them routable.

Network security
Out of the box, Docker has inter-container communication enabled (meaning the default is

```
--icc=true
```

); this means containers on a host can communicate with each other without any restrictions, which can potentially lead to denial-of-service attacks. Further, Docker controls the communication between containers and the wider world through the

```
--ip_forward
```

and

```
--iptables
```

flags. You should study the defaults of these flags and loop in your security team concerning company policies and how to reflect them in the Docker daemon setup. Also, check out the Docker security analysis Boyd Hemphill of StackEngine carried out.

Another network security aspect is that of on-the-wire encryption, which usually means TLS/SSL as per RFC 5246. Note, however, that at the time of this writing this aspect is rarely addressed; indeed, only two systems provide this out of the box: Weave uses NaCl and OpenVPN has a TLS-based setup. As Ive learned from Dockers security lead, Diogo Mnica, on-the-wire encryption will likely be available after v1.9.

Last but not least, check out Adrian Mouats Using Docker, which covers the network security aspect in great detail.

Tip

Automated Docker Security Checks

In order to automatically check against common security best practices around deploying Docker containers in production, I strongly recommend running The Docker Bench for Security.

[3]New Relic, for example, found the majority of the overall uptime of the containers, in one particular setup, in the low minutes; see also update here.

Post topics: Operations

Share: Tweet Share Share

THE O'REILLY APPROACH

Our Company

Teach/Write/Train

Careers

Community Partners

Diversity

SOLUTIONS

For Teams

For Enterprise

For Individuals

For Government

For Education

Marketing Solutions

SUPPORT

Contact Us

Privacy Policy

DOWNLOAD THE O'REILLY APP

Download on the App Store GET IT ON Google Play

Take O'Reilly online learning with you and learn anywhere, anytime on your phone or tablet. Download the app today and:

- Get unlimited access to books, videos, and live training
- Never lose your place—all your devices are synced
- Learn during your commute with online and offline access