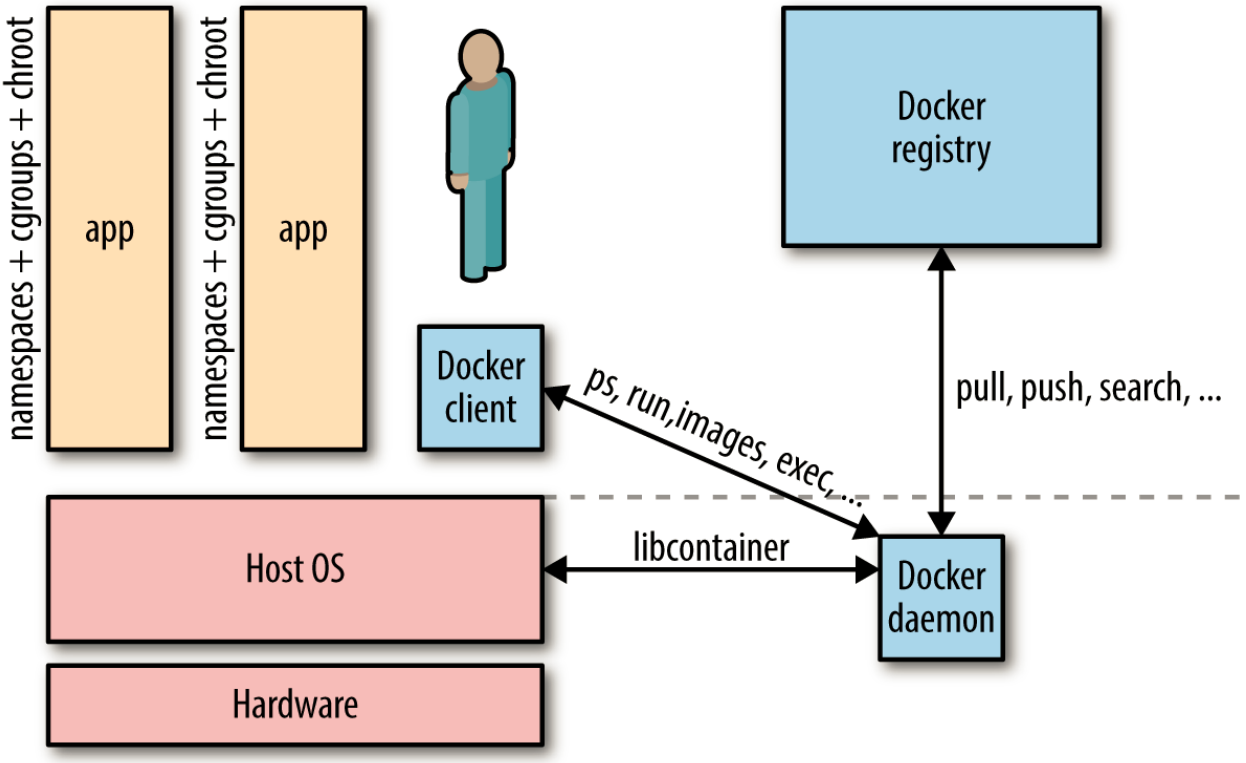
Docker Networking- Single Host

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.



The relationship between a host and containers is 1:*N*. This means that one host typically has several containers running on it.

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.

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

Bridge Mode Networking

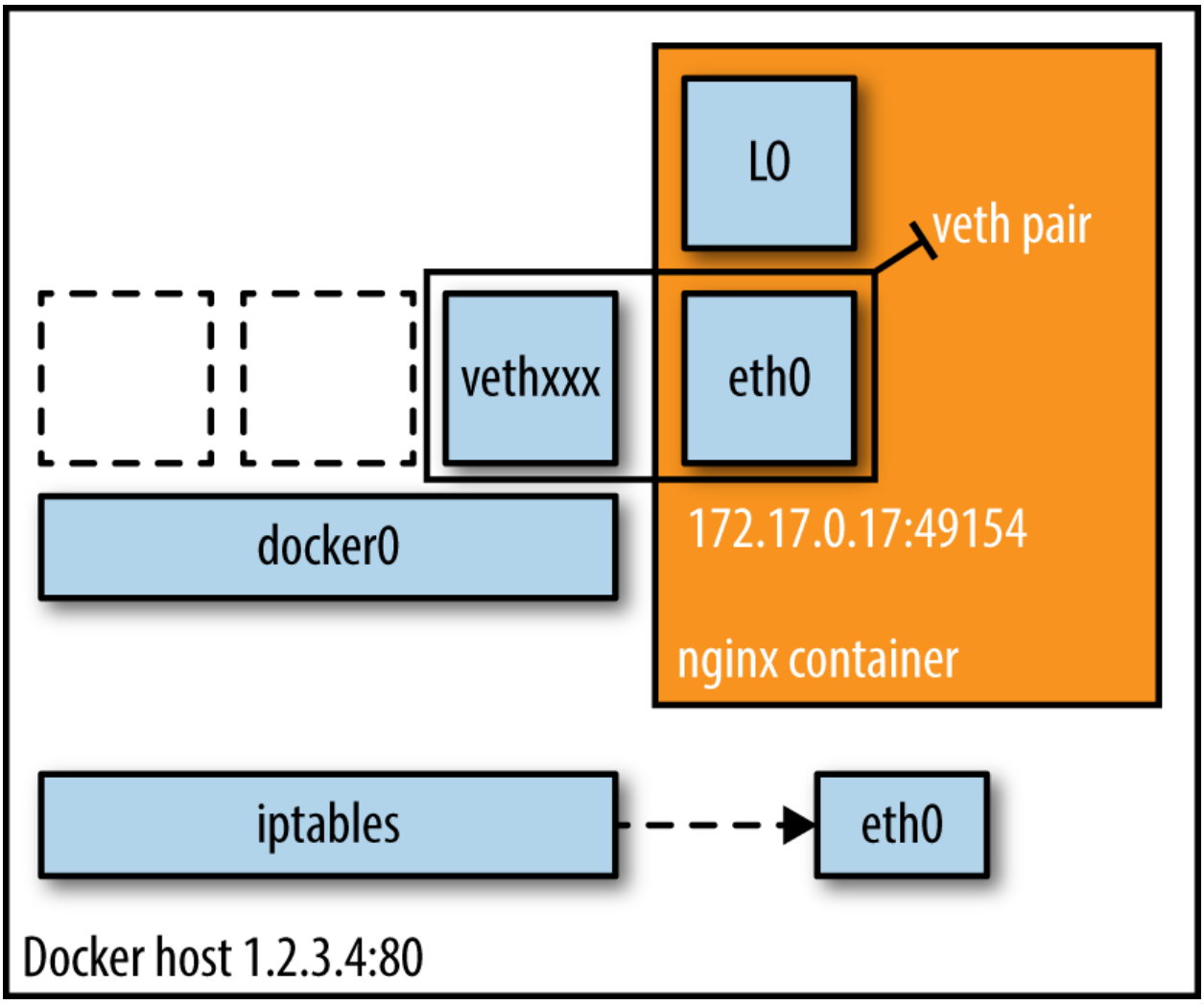
In this mode the Docker daemon creates `docker0`, a virtual Ethernet bridge that automatically forwards packets between any other network interfaces that are attached to it. By default, the daemon then connects all containers on a host to this internal network through creating a pair of peer interfaces,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



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

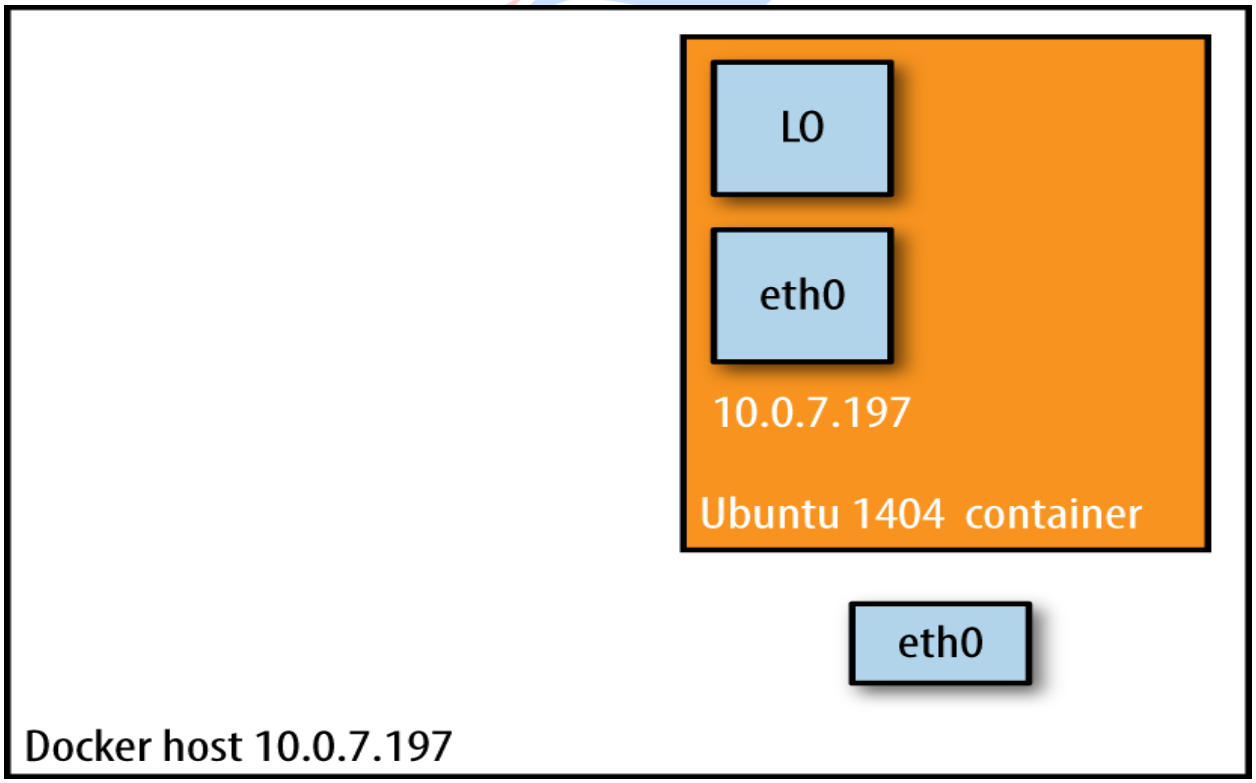
```
$ docker run -d --net=host ubuntu:14.04 tail -f /dev/null
$ ip addr | grep -A 2 eth0:
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 9001
qdisc mq state UP group default qlen 1000
  link/ether 06:58:2b:07:d5:f3 brd ff:ff:ff:ff:ff:ff
  inet **10.0.7.197**/22 brd 10.0.7.255 scope global dynamic
eth0

$ 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 default qlen 1000
  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.



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 group default
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 group default
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 doesn't configure it. Effectively, this turns off networking and is useful for two cases: either for containers that don't 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 configured—precisely as we would have hoped for.