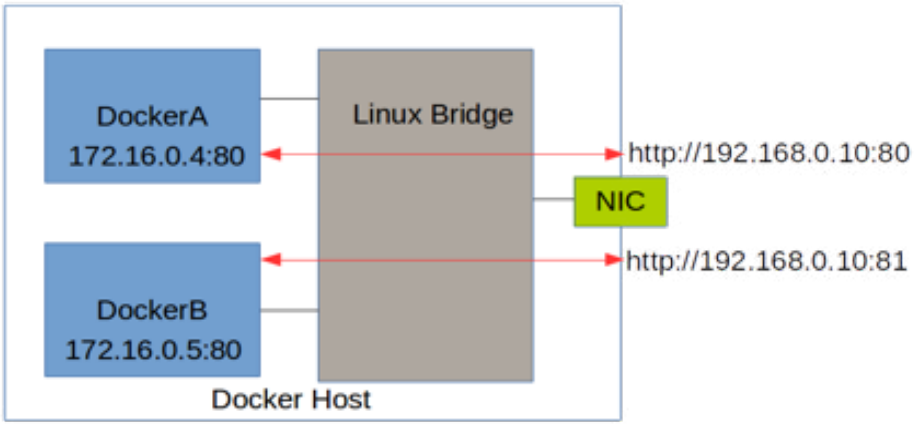
Docker Multi-Host Networking: Overlays to the Rescue

The Problem

The rise of Docker containers has driven the imagination of the IT world into a frenzy, with claims that they would revolutionize the way applications were deployed and services were consumed. And to some extent this was true – the appearance of Docker cloud services (AWS Elastic Container Service) and Docker cluster management and scheduling tools (Mesos, Kubernetes) have gone a long way to bringing the promise of container technology to fruition.

But the biggest issue to wide scale Docker deployments thus far I feel has been the somewhat cumbersome Docker networking implementation, particularly when it came to intra-container communication across multiple hosts. It was possible, but somewhat arcane to implement and orchestrate at a large scale (and there were a slew of tools developed to work around it – see Flannel, Weave, or Pipeworks).

The implementation depended on the use of local Linux bridges on each host to provide connectivity inside the host between the Dockers that resided on it...and used network port translation to provide inbound communication to the containers that the user needed to expose to the world. So if two containers on different hosts needed to talk to each other, it was necessary to determine the host ip and port that had been redirected to the internal Docker ip and port...and if that host died and the containers were restarted on another host, well then, redo that entire process. It also suffered from the issue of port contention in that only one Docker process could be translated to any individual port on the outside host.



However Docker purchased a startup known as SocketPlane earlier this year that had proposed a solution that should be familiar to anyone who has investigated the VMware NSX product...

The Solution

What the SocketPlane team proposed was using VXLAN encapsulation to build an overlay network of "tunnels" between the Docker hosts that would allow the container traffic to pass unmodified over the physical network. The original proposal involved using OpenVswitch (as a popular reference Linux VXLAN implementation at the time) on a host to connect the Docker internal network to other VXLAN tunnel endpoints (or VTEPs).

But after the acquisition Docker removed the OpenVswitch portion of the solution – the ability to do VXLAN off a Linux bridge had existed since 2012. So without the usage of additional software on the host, Docker could implement the solution natively within their main source code.

But there still exists a problem with VXLAN as an overlay network, and that lies in the control plane. In a standard physical network, endpoint discovery at layer 2 is handled through various mechanisms (arp and flooding primarily) that do not exist in an overlay network. How would a Docker host trying to pass traffic from a container it hosted to another container on a different host know where to send the packets?

One answer is to use a centralized control plane that contains all the destination MAC addresses of the containers mapped to the real hosts on which they sit. A second choice is to use multicast, with all the complexity that brings. And lastly, you can use a specialized protocol to advertise host/mac mappings such as BGP. Fortunately, for this purpose Docker decided to do a combination of the first and last options . Through the usage of a centralized key/value store (Etcd, Consul, or Zookeeper) the Docker hosts learn and advertise all the information needed to facilitate the magic at a management plane level. And the Docker hosts directly communicate host/mac mappings via the Serf gossip-based protocol to form a distributed control plane.

And so with the release of Docker engine 1.9 in November, Docker combined these two elements to provide a native solution to your multi-hosts woes via the usage of overlay networking.

Kicking the Tires

Ok enough with the background, let's play with the technology. For the purposes of getting us all on the same page I have created a vagrant setup that creates all the necessary components to get a lab up and running (an etcd server/Docker host, and one or more other Docker hosts). You can find it here. I'm going to reiterate some of the information found in the readme there at first, so bear with me.

After you clone the repo, you can change into the directory and run vagrant up. This (if you use the defaults) will create an etcd master server, and a single "minion" server that is configured as an etcd proxy. Both will get the latest version of Docker installed, and configured to be ready to go.

Now you can test the etcd service to see if it is up and running:

```
etcdctl cluster-health
```

You should see a similar response on any of the servers:

```
vagrant@etcdmaster:~$ etcdctl cluster-health
member 1dcf27d122fb08cb is healthy: got healthy result from
http://192.168.2.15:2379
cluster is healthy
```

These 3 default networks correspond to the the traditional (pre-1.9) docker networking models (bridged, null, and host mode).

```
vagrant@minion0:~$ docker network ls
NETWORK ID          NAME                DRIVER
93038eea0cd9        bridge              bridge
b83b588b89a3        none                null
d33237d314e0        host                host
```

These three default networks correspond to the traditional (pre-1.9) Docker networking models (bridged, null, and host-only).

You can also see the traditional docker0 interface has been created (which is tied to the "bridge" bridged network):

```
vagrant@minion0:~$ ip link
4: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue
state DOWN mode DEFAULT group default
    link/ether 02:42:ea:5b:3a:d8 brd ff:ff:ff:ff:ff:ff
```

If you create a test container, you can see the docker0 interface come up (transition from NO-CARRIER) and the virtual ethernet interface get created with docker0 as its master.

```
vagrant@minion0:~$ docker run -it busybox
vagrant@minion0:~$ ip link
4: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
state UP mode DEFAULT group default
    link/ether 02:42:ea:5b:3a:d8 brd ff:ff:ff:ff:ff:ff
7: veth3ac08be: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue master docker0 state UP mode DEFAULT group default
    link/ether 36:8e:5e:0d:7c:0f brd ff:ff:ff:ff:ff:ff
```

So it's worth talking about the virtual interface here or "veth". In the linux networking subsystems, a veth is, in reality, two virtual interfaces that are bonded together in a pair. Traffic that comes into one veth goes out the other, and vise-versa. So let's see where the other veth in the pair exists...

```
vagrant@minion0:~$ ethtool -S veth3ac08be
NIC statistics:
     peer_ifindex: 6
```

Note: ethtool is not installed by default on the virtual machines if you are following along. So issue **`sudo apt-get install ethtool`** if you want see this for yourself.

So our veth is bonded with a peer with index number 6. It doesn't show up on an ip link on the host, so let's check inside the Docker itself.

```
# ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
6: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:11:00:02 brd ff:ff:ff:ff:ff:ff
```

And there it is. So to summarize, a veth created inside the container (which it calls eth0) is linked with the veth in the host and that's how traffic moves in and out of the container. This will become important later.

So we can now describe the Docker network and get the information about it that shows the containers on the network, and their ip addresses.

```
vagrant@minion0:~$ docker network inspect bridge
[
    {
        “Name”: “bridge”,
        “Id”:
“93038eea0cd919e48fe5b3d47620a700837c49b8b43fc8e19b6f25ad528d83ff”,
        “Scope”: “local”,
        “Driver”: “bridge”,
        “IPAM”: {
            “Driver”: “default”,
            “Config”: [
                {
                    “Subnet”: “172.17.0.1/16”,
                    “Gateway”: “172.17.0.1”
                }
            ]
        },
        “Containers”: {

“e86330592614624de7af18a0ce5d914ceb99232c3da6a229d94ff82db7416e34”: {
                “EndpointID”:
“04f569ba9d321f7e06ffa563c7446c6cb3ae75d1dab26c3de2e7b39c9d7f3aee”,
                “MacAddress”: “02:42:ac:11:00:02”,
                “IPv4Address”: “172.17.0.2/16”,
                “IPv6Address”: “”
            }
        },
        “Options”: {
            “com.docker.network.bridge.default_bridge”: “true”,
            “com.docker.network.bridge.enable_icc”: “true”,
            “com.docker.network.bridge.enable_ip_masquerade”: “true”,
            “com.docker.network.bridge.host_binding_ipv4”: “0.0.0.0”,
            “com.docker.network.bridge.name”: “Docker0”,
            “com.docker.network.driver.mtu”: “1500”
        }
    }
]
```

See that part I highlighted? Notice the output of **docker ps:**

```
vagrant@minion0:~$ docker ps
CONTAINER ID        IMAGE               COMMAND             CREATED
STATUS              PORTS               NAMES
e86330592614        busybox             "sh"                29 minutes
ago      Up 29 minutes                          drunk_noyce
```

And you might also notice that the mac address listed in the output above matches with the output of the **ip list** command from inside the container. Cool!

But how do the containers get out to the world? The 172.17.0.0/16 ip address range is non-routable across the internet, and you may or may not have internal routing setup for that range.

If you look at the output of **docker network inspect** you can see the following line:
"com.docker.network.bridge.enable_ip_masquerade": "true"
and if you've ever set up a linux firewall you know what that means:

```
vagrant@minion0:~$ sudo iptables -L -t nat
...
Chain POSTROUTING (policy ACCEPT)
target     prot opt source               destination
MASQUERADE  all  --  172.17.0.0/16        anywhere
...
```

Add it all together, and it means that the traffic from the containers is network address translated (NAT) to the address of the Docker host.

One last bit on this section: with the new networking capability you can create additional host-only bridged networks for the purposes of segmentation within a Docker host. Let's try that.

```
vagrant@minion0:~$ docker network create bridgered
a2e2f2aedf083ef809e71cd1a100c700fa27b1df1ed3fa320f52308cb5f148f9
vagrant@minion0:~$ docker network ls
NETWORK ID          NAME                DRIVER
93038eea0cd9        bridge              bridge
b83b588b89a3        none                null
d33237d314e0        host                host
a2e2f2aedf08        bridgered           bridge
```

If I create another container and attach it to the new network, it will not be able to ping the ip address of the first container:

```
vagrant@minion0:~$ docker run -it --net bridgered busybox
/ # ip addr
8: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/16 scope global eth0
       valid_lft forever preferred_lft forever
/ # ping 172.17.0.2
PING 172.17.0.2 (172.17.0.2): 56 data bytes
^C
--- 172.17.0.2 ping statistics ---
14 packets transmitted, 0 packets received, 100% packet loss
```

"Ok," you say, "that's great and all, but where's the multi-host stuff? Have I been lied to?" Never fear dear reader, because as the old comedian used to say: I told you that story to tell you this one.

Multi-host Party

While the new easy multiple bridging additions are good stuff for single host environments, I said this post was about multi-host networking. And if you've been following along you have two vagrant virtual machines ready and waiting. So let's extend our examples to both machines.

First, let's create one of the new overlay networks. Pick either of your two Vms and issue the following:

```
vagrant@minion0:~$ docker network create --driver=overlay
--subnet=10.0.1.0/24 overlayred
f5ce773b3ff89e6e3998315fa0997b39c41dc0a05f8b6ae5d4ab78a9e42e8758
vagrant@minion0:~$ docker network ls
NETWORK ID          NAME                DRIVER
f5ce773b3ff8        overlayred          overlay
93038eea0cd9        bridge              bridge
b83b588b89a3        none                null
d33237d314e0        host                host
```

And now you're thinking that you should probably repeat that on the other host. No! If you go to the other host and docker network ls:

```
vagrant@etcdmaster:~$ docker network ls
NETWORK ID          NAME                DRIVER
f5ce773b3ff8        overlayred          overlay
4abd705bb316        bridge              bridge
a71b8b1e0f9d        none                null
e1332ccb2f6a        host                host
```

It already exists. How is this accomplished? Well if you remember we talked about the centralized key/value store (etcd in this case) that docker used as a management plane for overlay networks. And sure enough, if you start to poke around in the etcd store you will see the following:

```
vagrant@etcdmaster:~$ etcdctl ls docker/network/v1.0/network

docker/network/v1.0/network/
f5ce773b3ff89e6e3998315fa0997b39c41dc0a05f8b6ae5d4ab78a9e42e8758
```

Notice the first 12 characters of the "overlayred" network id match with the string in the etcdctl results. And if you were to retrieve the key/value pairs at that location you would see several familiar items.

```
vagrant@etcdmaster:~$ etcdctl get docker/network/v1.0/network/
f5ce773b3ff89e6e3998315fa0997b39c41dc0a05f8b6ae5d4ab78a9e42e8758

{"addrSpace":"GlobalDefault", "enableIPv6":false, "generic":{com.
docker.network.generic":{}},"id":"f5ce773b3ff89e6e3998315fa0997b-
39c41dc0a05f8b6ae5d4ab78a9e42e8758","ipamType":"default","ipamV-
4Config":"[{\"PreferredPool\":\"10.0.1.0/24\",\"SubPool\":\"\",\"Op-
tions\":null,\"Gateway\":\"\",\"AuxAddresses\":null}]","ipamV4In-
fo":"[{\"IPAMData\":\"{\\\"AddressSpace\\\":\\\"\\\",\\\"Gate-
way\\\":\\\"10.0.1.1/24\\\",\\\"Pool\\\":\\\"10.0.1.0/24\\\"}\",\"P
oolID\":\"GlobalDefault/10.0.1.0/24\"}]","name":"overlayred","net-
workType":"overlay","persist":true,"postIPv6":false}
```

And there are the configuration items we gave when we created the network (name, subnet, and network type). You can see most of this from the Docker command line, which I did on the other host because I could:

```
vagrant@minion0:~$ docker network inspect overlayred
[
    {
        “Name”: “overlayred”,
        “Id”:
“f5ce773b3ff89e6e3998315fa0997b39c41dc0a05f8b6ae5d4ab78a9e42e8758”,
        “Scope”: “global”,
        “Driver”: “overlay”,
        “IPAM”: {
            “Driver”: “default”,
            “Config”: [
                {
                    “Subnet”: “10.0.1.0/24”
                }
            ]
        },
        “Containers”: {},
        “Options”: {}
    }
]
```

Pretty slick. So let’s get some containers up.

```
vagrant@minion0:~$ docker run --net overlayred --name miniontest -it
busybox

vagrant@etcdmaster:~$ docker run --net overlayred --name mastertest -it
busybox
```

Note with the "**--net overlayred**" option I have assigned these containers to the overlayred network, and I have named each one of them. Let's look back at our etcd store and see what has happened.

```
vagrant@etcdmaster:~$ etcdctl ls /docker/network/v1.0/endpoint/
f5ce773b3ff89e6e3998315fa0997b39c41dc0a05f8b6ae5d4ab78a9e42e8758

docker/network/v1.0/endpoint/f5ce773b3ff89e6e3998315fa0997b39c41dc0a05f-
8b6ae5d4ab78a9e42e8758/a431e932facdce90c80cbbd3e04cb60fd6a3e32f17b6ca-
5f1e6e9990440c13be
docker/network/v1.0/endpoint/f5ce773b3ff89e6e3998315fa0997b39c41dc0a05f-
8b6ae5d4ab78a9e42e8758/821857d220175f1af18569e5c7109df7bae38f1405396eb-
fb7f98357a46a5a64
```

Hmm…two endpoints have been added to the network id we saw before. Let's get the key/values from one.

```
vagrant@etcdmaster:~$ etcdctl get /docker/network/v1.0/end-
point/f5ce773b3ff89e6e3998315fa0997b39c41dc0a05f8b6ae5d-
4ab78a9e42e8758/821857d220175f1af18569e5c7109df7bae38f1405396ebfb-
7f98357a46a5a64
{"anonymous":false,"ep_iface":{"addr":"10.0.1.2/24","dst-
Prefix":"eth","mac":"02:42:0a:00:01:02","routes":null,"s-
rcName":"veth2667354","v4PoolID":"GlobalDefault/10.0.1.0-
/24","v6PoolID":""},"exposed_ports":[],"generic":{"com.
docker.network.endpoint.exposedports":[],"com.docker.network.port-
map":[]},"id":"821857d220175f1af18569e5c7109df7bae38f1405396ebfb7f98357
a46a5a64","name":"miniontest","sandbox":"0a922e7b60a3aa360a6d58e625400-
2bf2b161e857914462964909105b13b9f82"}
```

And you can see the information about the container stored here, including its name, MAC address, and ip address. So the Docker hosts are using the centralized key/value server to store all the information about the network - and the hosts on that network - that is critical for communication.

If we look at the ip interfaces on one of the hosts, we can see some new additions.

```
vagrant@etcdmaster:~$ ip link
8: docker_gwbridge: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc
noqueue state UP mode DEFAULT group default
    link/ether 02:42:8e:3a:a6:99 brd ff:ff:ff:ff:ff:ff
10: vethdbf7407: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc no-
queue master docker_gwbridge state UP mode DEFAULT group default
    link/ether ba:ad:4c:53:7b:a5 brd ff:ff:ff:ff:ff:ff
```

It looks like a new bridge has been added (docker_gwbridge) and a veth has been added to it. Where is the other end of that veth? Let's run our ethtool command:

```
vagrant@etcdmaster:~$ ethtool -S vethdbf7407
NIC statistics:
     peer_ifindex: 9
```

On the container:

```
# ip addr
6: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
    link/ether 02:42:0a:00:01:03 brd ff:ff:ff:ff:ff:ff
    inet 10.0.1.3/24 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:aff:fe00:103/64 scope link
       valid_lft forever preferred_lft forever
9: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
    link/ether 02:42:ac:12:00:02 brd ff:ff:ff:ff:ff:ff
    inet 172.18.0.2/16 scope global eth1
       valid_lft forever preferred_lft forever
    inet6 fe80::42:acff:fe12:2/64 scope link
       valid_lft forever preferred_lft forever
```

It looks like that links with eth1 inside the container. But why does the container have two interfaces?

The answer is simple. When you use an overlay network for a container, it creates one interface (eth0 above) on the overlay network for intra-host communication over the overlay. But the container may still need access to the outside world, and routing that traffic through the overlay would require a logical device that could terminate the overlay traffic (act as a VTEP) and route that traffic out to the world. So Docker made the decision to create a "docker_gwbridge" on each host independently, and send all traffic headed outside of the overlay through the local host. If you rerun the iptables command from earlier you will also see another NAT/Masquerade entry has been added for the new docker_gwbridge network. The result is that the only traffic that crosses the overlay will be between containers on the same overlay network.

If we look at that eth0, however, we can see a few interesting points. First, it has an ip address within the 10.0.1.0/24 network, which we provided for our "overlayred" network. Secondly, it has an MTU size set to 1450 bytes, which is 50 bytes less than standard – and 50 bytes is exactly the size of a VXLAN header. So our eth0 interface within the container is configured to transit the overlay network. But where is its veth partner? Well, we know that eth0 has an ifindex of 6, so let's dig around.

Network Namespaces

To find it, we have to delve into network namespaces, as this is where the real magic occurs. Network namespaces are discrete and self-contained sets of interfaces and routing/forwarding tables that exist within a Linux system. The easiest analogy is VRF (virtual routing and forwarding) technology within most router platforms. It's a way to partition the network stack and make dedicated interfaces and forwarding tables that don't have the ability to interact with each other. If you read my previous blog post on Docker containers you may remember that containers have always used namespaces of different types to provide their compartmentalization – including networking. Well, with Docker 1.9, the use of network namespaces was extended to provide dedicated bridge namespaces for overlay networks.

You can see this by running this command:

```
vagrant@etcdmaster:~$ sudo ip netns list
de4b19bbc624
1-f5ce773b3f
```

(I have added a link from /var/run/docker/netns to /var/run/netns in the vagrant images for you. If you are doing this on a separate system, you will have to link these for the **ip netns** commands to read the Docker namespaces.)

You can see two namespaces exist. One of these is the namespace for the Docker container itself, and one is the namespace for the overlay network bridge.

```
vagrant@etcdmaster:~$ docker inspect 567a5 | grep "SandboxKey"
        "SandboxKey": "/var/run/docker/netns/de4b19bbc624",
```

So the top one **(de4b19bbc624)** is the namespace for the container. Let's look at the other one.

```
vagrant@etcdmaster:~$ sudo ip netns exec 1-f5ce773b3f ip addr
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN
group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
    inet 127.0.0.1/8 scope host lo
       valid_lft forever preferred_lft forever
    inet6 ::1/128 scope host
       valid_lft forever preferred_lft forever
2: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state
UP group default
    link/ether 76:d8:5b:0f:c1:05 brd ff:ff:ff:ff:ff:ff
    inet 10.0.1.1/24 scope global br0
       valid_lft forever preferred_lft forever
    inet6 fe80::40d2:8bff:fe47:7617/64 scope link
       valid_lft forever preferred_lft forever
5: vxlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue
master br0 state UNKNOWN group default
    link/ether c6:57:04:df:bf:35 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::c457:4ff:fedf:bf35/64 scope link
       valid_lft forever preferred_lft forever
7: veth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue mas-
ter br0 state UP group default
    link/ether 76:d8:5b:0f:c1:05 brd ff:ff:ff:ff:ff:ff
    inet6 fe80::74d8:5bff:fe0f:c105/64 scope link
       valid_lft forever preferred_lft forever
```

And there is the overlay network bridge. Notice the vxlan1 interface with a MTU of 1500, and the br0 and veth2 interfaces with MTUs of 1450. Let's check the link for veth2:

```
vagrant@etcdmaster:~$ sudo ip netns exec 1-f5ce773b3f ethtool -S veth2
NIC statistics:
     peer_ifindex: 6
```

And it's peer ifindex 6, which matches our eth0 in the output of the **ip link** command run inside the Docker above.

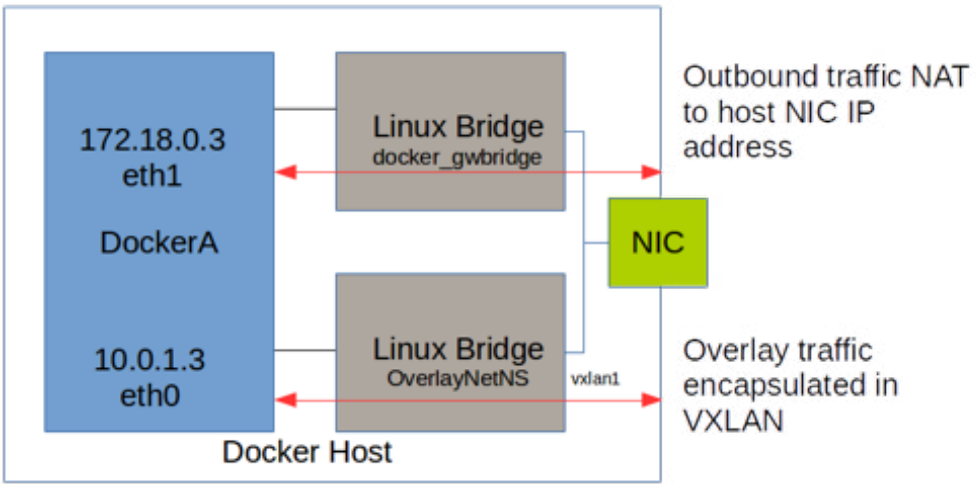
So the traffic for the overlay goes from eth0 in the Docker, out veth2 in the dedicated network namespace into the bridge, and then does what? Well, if we look at the forwarding database (FDB) for the bridge we see this entry:

```
vagrant@etcdmaster:~$ sudo ip netns exec 1-f5ce773b3f bridge fdb show
dev vxlan1
...
02:42:0a:00:01:02 dst 192.168.2.200 self permanent
```

192.168.2.200 is the ip address of our other server (minion0). What's that MAC address? If we go into the Docker on the minion0 server:

```
vagrant@minion0:~$ docker attach miniontest
/ # ip link
...
11: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue
    link/ether 02:42:0a:00:01:02 brd ff:ff:ff:ff:ff:ff
```

It's the MAC address of the docker's eth0. So the overlay bridge on etcdmaster knows the MAC address of the docker on the other server, and knows that to reach that MAC address you take the VXLAN tunnel to 192.168.2.200 - because it learned it via the Serf protocol.



Wow! That was a lot of in depth discussion, so let's do what we've all been waiting for and send some traffic over. One thing to note is that if you have given names to all the containers on the overlay network, you can reach them by name inside dockers on that overlay... much like the pre-existing Docker "link" feature, the container has the hostname/ip address mappings for each other container on the overlay injected into its host file. On the minion-test Docker (residing on the minion0 host):

```
/ # more /etc/hosts
10.0.1.2        62ba9269b985
127.0.0.1       localhost
::1     localhost ip6-localhost ip6-loopback
fe00::0 ip6-localnet
ff00::0 ip6-mcastprefix
ff02::1 ip6-allnodes
ff02::2 ip6-allrouters
10.0.1.3        mastertest
10.0.1.3        mastertest.overlayred
/ # ping mastertest
PING mastertest (10.0.1.3): 56 data bytes
64 bytes from 10.0.1.3: seq=0 ttl=64 time=0.576 ms
64 bytes from 10.0.1.3: seq=1 ttl=64 time=0.451 ms
64 bytes from 10.0.1.3: seq=2 ttl=64 time=0.436 ms
64 bytes from 10.0.1.3: seq=3 ttl=64 time=0.452 ms
64 bytes from 10.0.1.3: seq=4 ttl=64 time=0.445 ms
64 bytes from 10.0.1.3: seq=5 ttl=64 time=0.460 ms
^C
--- mastertest ping statistics ---
6 packets transmitted, 6 packets received, 0% packet loss
round-trip min/avg/max = 0.436/0.470/0.576 ms
```

And there we are - traffic between two Dockers on two separate hosts crossing the overlay network. You can see the VXLAN in action by going to the other server and doing some network captures.

```
vagrant@etcdmaster:~$ sudo tcpdump -i eth1 udp port 4789 -vvvn
tcpdump: listening on eth1, link-type EN10MB (Ethernet), capture size
65535 bytes
20:16:05.083210 IP (tos 0x0, ttl 64, id 17400, offset 0, flags [none],
proto UDP (17), length 134)
    192.168.2.200.46994 > 192.168.2.15.4789: [no cksum] VXLAN, flags
[I] (0x08), vni 256
IP (tos 0x0, ttl 64, id 12673, offset 0, flags [DF], proto ICMP (1),
length 84)
    10.0.1.2 > 10.0.1.3: ICMP echo request, id 2048, seq 150, length
64
```

This shows the outer VXLAN packet (from 192.168.2.200 to 192.168.2.15) on UDP port 4789, and inside that packet is an IP ICMP packet from 10.0.1.2 to 10.0.1.3.

Multi-Tenancy and Wrapping Up

Let's see what happens if we try and make another overlay with the same ip address range – similar to what might happen in a multi-tenancy environment.

```
vagrant@etcdmaster:~$ docker network create --driver=overlay --sub-
net=10.0.1.0/24 overlayblue
d43e9dbb333d2829b427271f62634830c97519c6e7f4f64ce416dcf5944c0296
vagrant@etcdmaster:~$ docker network ls
NETWORK ID          NAME                DRIVER
d43e9dbb333d        overlayblue         overlay
f5ce773b3ff8        overlayred          overlay
```

And it created fine…both are listed. Let's make new containers on the network:

```
vagrant@etcdmaster:~$ docker run -it --net overlayblue --name master-
blue busybox

vagrant@minion0:~$ docker run --net overlayblue --name minionblue -it
busybox
/ # ping masterblue
PING masterblue (10.0.1.6): 56 data bytes
64 bytes from 10.0.1.6: seq=0 ttl=64 time=0.463 ms
64 bytes from 10.0.1.6: seq=1 ttl=64 time=0.464 ms
64 bytes from 10.0.1.6: seq=2 ttl=64 time=0.440 ms
^C
--- masterblue ping statistics ---
3 packets transmitted, 3 packets received, 0% packet loss
round-trip min/avg/max = 0.440/0.455/0.464 ms
/ # ping 10.0.1.3
PING 10.0.1.3 (10.0.1.3): 56 data bytes
^C
--- 10.0.1.3 ping statistics ---
2 packets transmitted, 0 packets received, 100% packet loss
```

We can see there is network segregation (can't talk between the two overlay networks), but they are sharing the same subnet and not allowing duplicate ip addresses. The two overlays will use a separate VNI (VXLAN network identifier) to show each discrete network segment.

What about routed overlays? After all, our two hosts are sitting in the same 192.168.2.0/24 subnet. In testing, it seems to work fine across a router as long as the VXLAN ports are allowed, and all hosts can reach the centralized key/value store. If you are interested in playing around with layer 3, I've created another vagrant setup that mirrors the one we've been using, but has a Vyos (Vyatta) router in the middle. You can find it here.

That about does it for this discussion of the new Docker networking features. While I went quite in depth in some parts of this, I hope that you have found it useful and informative. It is my belief that these new networking features will help to grow the adoption of container technology far beyond its current state.

Don Mills
Managing Consultant

About SingleStone

Founded in 1997, SingleStone is a consulting firm specializing in customer experience solutions, spanning technology, strategy, business processes and culture. We help our clients improve their interactions and communications with customers by combining these disciplines to create, implement and support end-to-end customer experiences that benefit both business and humanity.

Connect With Us

4101 Cox Road, Suite 350
Glen Allen, VA 23060

804.648.0600
SingleStoneConsulting.com

singlestone℠