NETWORK SECURITY (HTTPS://NEUVECTOR.COM/CATEGORY/NETWORK-SECURITY/)

How Docker Swarm Container Networking Works – Under the Hood (https://neuvector.com/network-security/docker-swarm-container-networking/)

By Gary Duan, CTO, NeuVector

Docker 1.12 is a release loaded with a lot of great features. With built-in orchestration and by removing dependencies on the external KV store, Docker Swarm allows DevOps to quickly deploy a multi-host docker cluster that "just works." Although not without controversies, when compared to Kubernetes, Docker Swarm's ease-of-use is one of it's most cited advantages. Perhaps in response, Kubernetes simplified its cluster bootstrapping process with the introduction of kubeadm in its 1.4 release.

In this post on Swarm container networking I'll assume you have mastered the basics of creating a swarm and starting services. I won't be covering general concepts of managing swarm nodes and services. Instead, the focus will be on explaining how Docker uses various Linux tools to virtualize multi-host overlay networks. You can also see my more recent post on Kubernetes networking (https://neuvector.com/network-security/kubernetes-networking/) basics, or this detailed post on Advanced Kubernetes Networking (https://neuvector.com/network-security/advanced-kubernetes-networking/) which includes how to deploy multiple networks.

Deployment

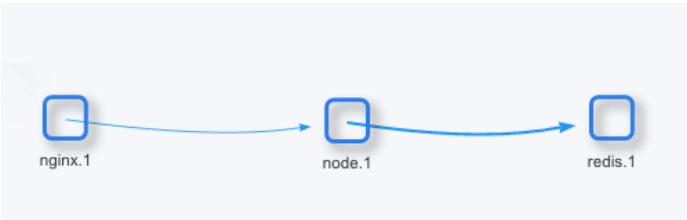
First, let's create an overlay network and deploy our containers. Our docker swarm cluster has two nodes. I will create three services, each has one running instance.

```
docker network create --opt encrypted --subnet 100.0.0.0/24 -d overlay net1

docker service create --name redis --network net1 redis
docker service create --name node --network net1 nvbeta/node
docker service create --name nginx --network net1 -p 1080:80 nvbeta/swarm_nginx
```

The above command creates a typical 3-tier application. The 'nginx' container as a load balancer redirects traffic to 'node' container as a web server, then the 'node' container accesses the 'redis' database and displays the result to the user. For simplicity, only one web server is created behind the load balancer.

This is the logical view of the application.



Networks

Let's look at the networks created by Docker Swarm,

```
$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
cac91f9c60ff        bridge              bridge              local
b55339bbfab9        docker_gwbridge     bridge              local
fe6ef5d2e8e8        host                host                local
f1nvcluv1xnf        ingress             overlay             swarm
8vty8k3pejm5        net1                overlay             swarm
893a1bbe3118        none                null                local
```

net1:

> *This is the overlay network we create for east-west communication between containers.*

docker_gwbridge:

> *This is the network created by Docker. It allows the containers to connect to the host that it is running on.*

ingress:

> *This is the network created by Docker. Docker swarm uses this network to expose services to the external network and provide the routing mesh.*

net1

Because all services are created with the "–network net1" option, each of them must have one interface connecting to the 'net1' network.

Let's use node 1 as an example. There are two containers deployed on node 1 by Docker Swarm.

```
$ docker ps
CONTAINER ID        IMAGE               COMMAND                  CREATED             STATUS              PORTS
NAMES
eb03383913fb        nvbeta/node:latest   "nodemon /src/index.j"   2 hours ago         Up 2 hours          8888/tcp
node.1.10yscmxtoymkvs3bdd4z678w4
434ce2679482        redis:latest         "docker-entrypoint.sh"   2 hours ago         Up 2 hours          6379/tcp
redis.1.1a2l4qmvg887xjpfklp4d6n7y
```

By creating a symbolic link to the docker netns folder, we can find out all network namespaces on node 1.

```
$ cd /var/run
$ sudo ln -s /var/run/docker/netns netns
$ sudo ip netns
be663feced43
6e9de12ede80
2-8vty8k3pej
1-f1nvcluv1x
72df0265d4af
```

Comparing the namespace names with the Docker swarm network IDs, we can guess that namespace '*2-8vty8k3pej*' is used for 'net1' network, whose ID is *8vty8k3pejm5*. This can be confirmed by comparing interfaces in the namespace and containers.

Interface list in the containers:

```
$ docker exec node.1.10yscmxtoymkvs3bdd4z678w4 ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
11040: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:65:00:00:03 brd ff:ff:ff:ff:ff:ff
11042: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:ac:12:00:04 brd ff:ff:ff:ff:ff:ff
```

```
$ docker exec redis.1.1a2l4qmvg887xjpfklp4d6n7y ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
11036: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:65:00:00:08 brd ff:ff:ff:ff:ff:ff
11038: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:ac:12:00:03 brd ff:ff:ff:ff:ff:ff
```

Interface list in the namespace:

```
$ sudo ip netns exec 2-8vty8k3pej ip link
1: lo: <LOOPBACK,UP,LOWER_UP> mtu 65536 qdisc noqueue state UNKNOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
2: br0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP mode DEFAULT group default
    link/ether 22:37:32:66:b0:48 brd ff:ff:ff:ff:ff:ff
11035: vxlan1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UNKNOWN mode DEFAULT group default
    link/ether 2a:30:95:63:af:75 brd ff:ff:ff:ff:ff:ff
11037: veth2: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UP mode DEFAULT group default
    link/ether da:84:44:5c:91:ce brd ff:ff:ff:ff:ff:ff
11041: veth3: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue master br0 state UP mode DEFAULT group default
    link/ether 8a:f9:bf:c1:ec:09 brd ff:ff:ff:ff:ff:ff
```

Note that '*br0*' is the Linux Bridge where all the interfaces are connected to; '*vxlan1*' is the VTEP interface for VXLAN overlay network. For each veth pair that Docker creates for the container, the device inside the container always has an ID number which is 1 number smaller than the device ID of the other end. So, *veth2* (11037) in the namespace is connected to the *eth0* (11036) in the redis container; *veth3* (11041) in the namespace is connected to the *eth0* (11040) in the node container.

We can be certain now that namespace '*2-8vty8k3pej*' is used for 'net1' overlay network. We can draw the network diagram on node 1 based on what we have learned.

```
               node 1


  +-----------+      +-----------+
  |  nodejs   |      |   redis   |
  |           |      |           |
  +---------+-+      +---------+-+
            |                  |
            |                  |
            |                  |
            |                  |
       +----+------------------+-------+ net1
        101.0.0.3          101.0.0.8
        101.0.0.4(vip)     101.0.0.2(vip)
```

docker_gwbridge

We can compare the 'redis' and 'node' container's interfaces with the interfaces list on the host.

Interface list on the host,

```
$ ip link
...
4: docker_gwbridge: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP mode DEFAULT group default
    link/ether 02:42:24:f1:af:e8 brd ff:ff:ff:ff:ff:ff
5: docker0: <NO-CARRIER,BROADCAST,MULTICAST,UP> mtu 1500 qdisc noqueue state DOWN mode DEFAULT group default
    link/ether 02:42:e4:56:7e:9a brd ff:ff:ff:ff:ff:ff
11039: veth97d586b: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker_gwbridge state UP mode DEFAULT
group default
    link/ether 02:6b:d4:fc:8a:8a brd ff:ff:ff:ff:ff:ff
11043: vethefdaa0d: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker_gwbridge state UP mode DEFAULT
group default
    link/ether 0a:d5:ac:22:e7:5c brd ff:ff:ff:ff:ff:ff
10876: vethceaaebe: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue master docker_gwbridge state UP mode DEFAULT
group default
    link/ether 3a:77:3d:cc:1b:45 brd ff:ff:ff:ff:ff:ff
...
```

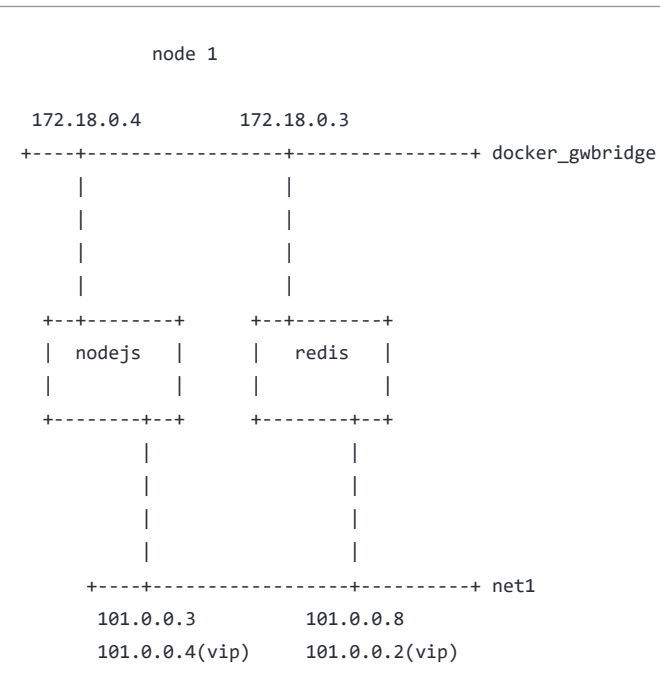
Recall the device ID relation of the veth pair, we know that, *veth97d586b* (11039) on the host is connected to the *eth1* (11038) in the redis container; *vethefdaa0d* (11043) on the host is connected to the *eth1* (11042) in the node container.

These two interfaces on the host are on the docker_gwbridge bridge,

```
$ brctl show
bridge name             bridge id               STP enabled         interfaces
docker0                 8000.0242e4567e9a       no
docker_gwbridge         8000.024224f1afe8       no                  veth97d586b
                                                                    vethceaaebe
                                                                    vethefdaa0d
```

The docker_gwbridge on each host is very much like the docker0 bridge in the single-host Docker environment. Each container has a leg connecting to it and it's reachable from the host that the container is running on.

Node 1's network diagram can be updated, with host networking added.



However, unlike the docker0 bridge, it is not used to connect to the external network. For Docker Swarm services that publishes ports (with the -p option), Docker creates a dedicated 'ingress' network for it.

ingress

Again the network namespace list and Docker Swarm network list on node 1,

```
$ sudo ip netns
be663feced43
6e9de12ede80
2-8vty8k3pej
1-f1nvcluv1x
72df0265d4af

$ docker network ls
NETWORK ID          NAME                DRIVER              SCOPE
cac91f9c60ff        bridge              bridge              local
b55339bbfab9        docker_gwbridge     bridge              local
fe6ef5d2e8e8        host                host                local
f1nvcluv1xnf        ingress             overlay             swarm
8vty8k3pejm5        net1                overlay             swarm
893a1bbe3118        none                null                local
```

Cleary, *1-f1nvcluv1x* is the namespace of 'ingress' network, but what is the *72df0265d4af* namespace used for?

Let's first look at what interfaces are in the namespace.

```
$ sudo ip netns exec 72df0265d4af 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
10873: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1450 qdisc noqueue state UP group default
    link/ether 02:42:0a:ff:00:03 brd ff:ff:ff:ff:ff:ff
    inet 10.255.0.3/16 scope global eth0
       valid_lft forever preferred_lft forever
    inet6 fe80::42:aff:feff:3/64 scope link
       valid_lft forever preferred_lft forever
10875: eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc noqueue state UP group default
    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
```

eth1 (10875) is paired with *vethceaaebe* (10876) on the host; we can also find out that *eth0* is connected to 'ingress' overlay network. The interface setup looks like a container's network namespace.

This can proved by inspecting Docker Swarm 'ingress' and 'docker_gwbridge' network on node 1.

```
$ docker network inspect ingress
[
    {
        "Name": "ingress",
        "Id": "f1nvcluv1xnfa0t2lca52w69w",
        "Scope": "swarm",
        "Driver": "overlay",
        ....
        "Containers": {
            "ingress-sbox": {
                "Name": "ingress-endpoint",
                "EndpointID": "3d48dc8b3e960a595e52b256e565a3e71ea035bb5e77ae4d4d1c56cab50ee112",
                "MacAddress": "02:42:0a:ff:00:03",
                "IPv4Address": "10.255.0.3/16",
                "IPv6Address": ""
            }
        },
        ....
    }
]

$ docker network inspect docker_gwbridge
[
    {
        "Name": "docker_gwbridge",
        "Id": "b55339bbfab9bdad4ae51f116b028ad7188534cb05936bab973dceae8b78047d",
        "Scope": "local",
        "Driver": "bridge",
        ....
        "Containers": {
            ....
            "ingress-sbox": {
                "Name": "gateway_ingress-sbox",
                "EndpointID": "0b961253ec65349977daa3f84f079ec5e386fa0ae2e6dd80176513e7d4a8b2c3",
                "MacAddress": "02:42:ac:12:00:02",
                "IPv4Address": "172.18.0.2/16",
                "IPv6Address": ""
            }
        },
        ....
    }
]
```

The endpoints' MAC/IP addresses on these networks match the interface MAC/IP addresses in the namespace *72df0265d4af*. This namespace is for the hidden container named '**ingress-sbox,'** which has one leg on the host network and another leg on 'ingress' network.

One of the major features of Docker Swarm is the 'routing mesh' for containers that publish ports. No matter which node the container instance is actually running on, you can access it through any node. How is it done? Let's dig deeper into this hidden container.

In our application, it's the 'nginx' service that publishes port 80 and maps to port 1080 on the host, but the 'nginx' container is not running on node 1.

Still on node 1,

```
$ sudo iptables -t nat -nvL
...
...
Chain DOCKER-INGRESS (2 references)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DNAT       tcp  --  *      *       0.0.0.0/0            0.0.0.0/0            tcp dpt:1080 to:172.18.0.2:1080
 176K   11M RETURN     all  --  *      *       0.0.0.0/0            0.0.0.0/0

$ sudo ip netns exec 72df0265d4af iptables -nvL -t nat
...
...
Chain PREROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    9   576 REDIRECT   tcp  --  *      *       0.0.0.0/0            0.0.0.0/0            tcp dpt:1080 redir ports 80
...
...
Chain POSTROUTING (policy ACCEPT 0 packets, 0 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 DOCKER_POSTROUTING  all  --  *      *       0.0.0.0/0            127.0.0.11
   14   896 SNAT       all  --  *      *       0.0.0.0/0            10.255.0.0/16        ipvs to:10.255.0.3
```

As you can see, iptables rules redirect the traffic to port 1080 to port 80 in the hidden container 'ingress-sbox.' Then the POSTROUTING chain puts the packets on IP 10.255.0.3, which is the IP address of interface on 'ingress' network.

Notice 'ipvs' in the SNAT rule. '**ipvs**' is a load balancer implementation in the Linux kernel. It's a little-known tool that has been in the kernel for 16 years.

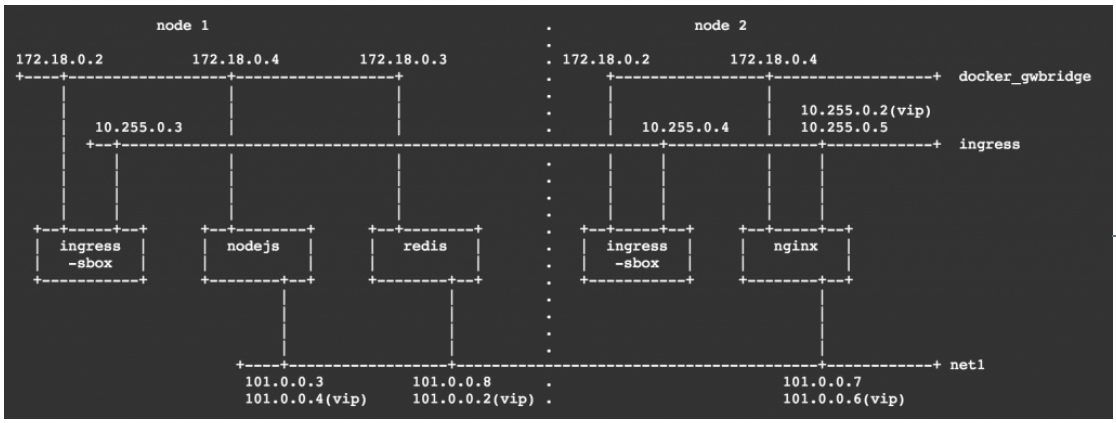
```
$ sudo ip netns exec 72df0265d4af iptables -nvL -t mangle
Chain PREROUTING (policy ACCEPT 144 packets, 12119 bytes)
 pkts bytes target     prot opt in     out     source               destination
   87  5874 MARK       tcp  --  *      *       0.0.0.0/0            0.0.0.0/0            tcp dpt:1080 MARK set 0x12c
...
...
Chain OUTPUT (policy ACCEPT 15 packets, 936 bytes)
 pkts bytes target     prot opt in     out     source               destination
    0     0 MARK       all  --  *      *       0.0.0.0/0            10.255.0.2           MARK set 0x12c
...
...
```

The iptables rule mark the flow as 0x12c (= 300), then here is how ipvs is configured,

```
$ sudo ip netns exec 72df0265d4af ipvsadm -ln
IP Virtual Server version 1.2.1 (size=4096)
Prot LocalAddress:Port Scheduler Flags
  -> RemoteAddress:Port           Forward Weight ActiveConn InActConn
FWM  300 rr
  -> 10.255.0.5:0                 Masq    1      0          0
```

10.255.0.5 is the IP of the 'nginx' container on the other node. It's the only backend server for the load balancer.

With everything coming together, we can update our network connection view (click to expand).



(https://neuvector.com/wp-content/uploads/2017/01/swarm_network_view.png)

Summary

There's a lot of cool things happening under the hood when using Docker Swarm container networking. This makes it easy to deploy more complex production applications on multi-host networks. The use of overlay networks makes it easy to scale and move containers across nodes and even across cloud environments. However, it's always useful to understand what's happening under the hood, so that any connectivity issues can be more easily debugged. The increase in 'east-west (https://neuvector.com/blog/securing-east-west-traffic-in-container-based-data-center/)' traffic between containers also means that new approaches to visibility and security are required.

[For an overview of Kubernetes networking and security, download ***The Ultimate Kubernetes Security Guide***]

About the Author: Gary Duan

Gary is the Co-Founder and CTO of NeuVector. He has over 15 years of experience in networking, security, cloud, and data center software. He was the architect of Fortinet's award winning DPI product and has managed development teams at Fortinet, Cisco and Altigen. His technology expertise includes IDS/IPS, OpenStack, NSX and orchestration systems. He holds several patents in security and data center technology.

About the Author

NeuVector delivers an application-aware container network security solution. The NeuVector containers deploy easily in minutes and discover running services and applications. A security policy is automatically created and updated when containers launch, scale up or scale down. NeuVector detects container threats, violations, and vulnerabilities.

By NeuVector
Tags: Container Networking (https://neuvector.com/tag/container-networking/), Docker Security (https://neuvector.com/tag/docker-security/)
One Comment