Docker Multi-host Overlay Networking with Etcd

2015-11-09

Docker has released its newest version v1.9 (see details) on November 3, 2015. This big release put Swarm and multi-host networking into production-ready status. This blog illustrates the configuration and a few evaluations of Docker multi-host overlay networking.

Multi-host Networking

Multi-host Networking was announced as part of experimental release in June, 2015, and turns to stable release of Docker Engine this month. There are already several Multi-host networking solutions for docker, such as Calico and Flannel. Docker multi-host networking uses VXLAN-based solution with the help of `libnetwork` and `libkv` library. So the `overlay` network requires a valid key-value store service to exchange informations between different docker engines. Docker implements a built-in VXLAN-based overlay network driver in `libnetwork` library to support a wide range virtual network between multiple hosts.

Prerequisite

Environment Preparation

Before using Docker overlay networking, check the version of docker with `docker -v` to confirm that docker version is no less than v1.9. In this blog I prepare an environment with two Linux nodes (node1/node2) with IP 192.168.236.130/131 and connect them physically or virtually, and confirm they have network access to each other.

ownload and run etcd, replace {node} with node0/1 seperately. We need at

least two etcd node since the new version of etcd cannot run on single node.

Download and run etcd

```
curl -L  https://github.com/coreos/etcd/releases/download/v2.2.1/etcd-v2.2.1-linux-amd64.tar.gz -o e
tar xzvf etcd-v2.2.1-linux-amd64.tar.gz
cd etcd-v2.2.1-linux-amd64
./etcd -name {node} -initial-advertise-peer-urls http://{NODE_IP}:2380 \
  -listen-peer-urls http://0.0.0.0:2380 \
  -listen-client-urls http://0.0.0.0:2379,http://127.0.0.1:4001 \
  -advertise-client-urls http://0.0.0.0:2379 \
  -initial-cluster-token etcd-cluster \
  -initial-cluster node1=http://192.168.236.130:2380,node2=http://192.168.236.131:2380 \
  -initial-cluster-state new
```

Start Docker Daemon With Cluster Parameters

Docker Engine daemon should be started with cluster parameters `--cluster-store` and `--cluster-advertise`, thus all Docker Engine running on different nodes could communicate and cooperate with each other. Here we need to set `--cluster-store` with Etcd service host and port and `--cluster-advertise` with IP and Docker Daemon port on this node. Stop current docker daemon and start with new params.

On node1:

Run Docker daemon with cluster params

```
sudo service docker stop
sudo /usr/bin/docker daemon -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --cluster-store=etcd
```

On node2:

Run Docker daemon with cluster params

```
sudo service docker stop
sudo /usr/bin/docker daemon -H tcp://0.0.0.0:2375 -H unix:///var/run/docker.sock --cluster-store=etcd
```

All preparations are done until now.

Create Overlay Network

On either node, we can execute `docker network ls` to see the network configuration of Docker. Here's the example of node1:

Docker network configuration

```
docker@node1:~
NETWORK ID          NAME                DRIVER
80a36a28041f        bridge              bridge
6b7eab031544        none                null
464fe03753fb        host                host
```

Then we also use `docker network` command to create a new overlay network.

Docker network configuration

```
docker@node1:~
904f9dc335b0f91fe155b26829287c7de7c17af5cfeb9c386a1ccf75c42cd3eb
```

Wait for a minute and we can see the output of this command is the ID of this overlay network. Then execute `docker network ls` on either node:

Docker network configuration

```
docker@node1:~
NETWORK ID          NAME                DRIVER
904f9dc335b0        myapp               overlay
80a36a28041f        bridge              bridge
6b7eab031544        none                null
464fe03753fb        host                host
52e9119e18d5        docker_gwbridge     bridge
```

On both node1 and node2, two network `myapp` and `docker_gwbridge` are added with type `overlay` and `bridge` seperately. Thus `myapp` represents the overlay network associated with `eth0` in containers, and `docker_gwbridge` represents the bridge network connecting Internet associated with `eth1` in containers.

Create Containers With Overlay Network

On node1:

Docker network configuration

And on node2:

Docker network configuration

Then test the connection between two containers. On node1, execute:

Docker networks

```
docker@node1:~/etcd-v2.0.9-linux-amd64
eth0      Link encap:Ethernet  HWaddr 02:42:0a:00:00:02
          inet addr:10.0.0.2  Bcast:0.0.0.0  Mask:255.255.255.0
          inet6 addr: fe80::42:aff:fe00:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1450  Metric:1
          RX packets:5475264 errors:0 dropped:0 overruns:0 frame:0
          TX packets:846008 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:7999457912 (7.9 GB)  TX bytes:55842488 (55.8 MB)

eth1      Link encap:Ethernet  HWaddr 02:42:ac:12:00:02
          inet addr:172.18.0.2  Bcast:0.0.0.0  Mask:255.255.0.0
          inet6 addr: fe80::42:acff:fe12:2/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:12452 errors:0 dropped:0 overruns:0 frame:0
          TX packets:6883 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:22021017 (22.0 MB)  TX bytes:376719 (376.7 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:0
          RX bytes:0 (0.0 B)  TX bytes:0 (0.0 B)
```

Here we can see two NICs in container with IP 10.0.0.2 and 172.18.0.2. `eth0` connects to the overlay network and `eth1` connects to docker_gwbridge. Thus the container will both have access to containers on other host as well as Google. Run the same command on node2 and we can see the IP of `eth0` in worker-2 is 10.0.0.3, which is assigned continuously.

Then test the connections between worker-1 and worker-2, execute command on node1:

Docker network configuration

```
docker@node1:~
PING 10.0.0.3 (10.0.0.3) 56(84) bytes of data.
64 bytes from 10.0.0.3: icmp_seq=1 ttl=64 time=0.735 ms
64 bytes from 10.0.0.3: icmp_seq=2 ttl=64 time=0.581 ms
64 bytes from 10.0.0.3: icmp_seq=3 ttl=64 time=0.444 ms
64 bytes from 10.0.0.3: icmp_seq=4 ttl=64 time=0.447 ms

--- 10.0.0.3 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3000ms
rtt min/avg/max/mdev = 0.444/0.551/0.735/0.122 ms
```

Performance Tests

I did a simple performance test between two containers with `iperf`, and here is the result.

First I tested the native network performance between node1 and node2:

```
docker@node2:~# iperf -c 192.168.236.130
------------------------------------------------------------
Client connecting to 192.168.236.130, TCP port 5001
TCP window size:  136 KByte (default)
------------------------------------------------------------
[  3] local 192.168.236.131 port 36910 connected with 192.168.236.130
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  2.59 GBytes  2.22 Gbits/sec
```

Then network performance between worker-1 and worker-2:

```
root@3f8bc51fb458:~# iperf -c 10.0.0.2
------------------------------------------------------------
Client connecting to 10.0.0.2, TCP port 5001
TCP window size: 81.0 KByte (default)
------------------------------------------------------------
[  3] local 10.0.0.3 port 48096 connected with 10.0.0.2 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-10.0 sec  1.84 GBytes  1.58 Gbits/sec
```

The overlay network performance is a bit worse than native. It's also a little worse than Calico, which is almost the same as native performance. Since Calico uses a pure 3-Layer protocol and Docker Multi-host Overlay

Network uses VXLAN solution (MAC on UDP), Calico does make sense to gain a better performance.

VXLAN Technology

Virtual Extensible LAN (VXLAN) is a network virtualization technology that attempts to ameliorate the scalability problems associated with large cloud computing deployments. It uses a VLAN-like encapsulation technique to encapsulate MAC-based OSI layer 2 Ethernet frames within layer 4 UDP packets. Open vSwitch is a former implementation of VXLAN, but Docker Engine implements a built-in VXLAN driver in libnetwork.

For more VXLAN details, you can see its official RFC and a white paper from EMulex. I'd like to post another blog to have more detailed discussion on VXLAN Technology.

References

[1] Docker Multi-host Networking Post:
http://blog.docker.com/2015/11/docker-multi-host-networking-ga/
[2] Docker Network Docs:
http://docs.docker.com/engine/userguide/networking/dockernetworks/
[3] Get Started Overlay Network for Docker:
https://docs.docker.com/engine/userguide/networking/get-started-overlay/
[4] Docker v1.9 Announcemount:
https://blog.docker.com/2015/11/docker-1-9-production-ready-swarm-multi-host-networking/
[5] VXLAN Official RFC: https://datatracker.ietf.org/doc/rfc7348/
[6] VXLAN White Paper: https://www.emulex.com/artifacts/d658610a-d3b6-457c-bf2d-bf8d476c6a98/elx_wp_all_VXLAN.pdf