Konrad Scherer

Docker Multi Host Networking

Introduction

I recently did a presentation at work covering the basics of getting docker container on different hosts to talk to one another. This was motivated because I wanted to understand all the available strange networking options and why Kubernetes choose the one network per pod model as the default.

Docker networking breaks many of the current assumptions about networking. A modern server can easily run 100+ containers and a datacenter rack can hold 80+ servers. If the networking model is one IP per container, that implies 100+ IPs per machine and 1000s per rack. Ephemeral containers with a short lifespan means that the network has to react quickly.

Of course there are competing container networking standards: CNM (libnetwork from Docker) and CNI (CoreOS and Kubernetes). Beyond the supported network models in Docker there is also a docker network plugin ecosystem with various vendors providing special integration with their gear.

Bridge Mode

Let's start simple with the default bridge mode. Docker creates a linux bridge and veth per container. By default containers can access external network but external network cannot access container. This is the safe default. To allow external access to a container, host ports are forwarded to container ports. IPTables rules to prevent inter container communication. This functionality works with older kernels

Bridge mode example

```
> docker run --detach --publish 1234:1234 ubuntu:16.04 sleep infinity

# docker0 is the bridge, veth is connected to the docker0 bridge
> ip addr
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    inet 128.224.56.107/24 brd 128.224.56.255 scope global eth0
3: docker0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500
    inet 172.17.0.1/16 scope global docker0
8: vethea44ea7@if7: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 master docker0 state UP

# iptables rules for packet forwarding
> iptables -L
Chain FORWARD (policy DROP)
target     prot opt source               destination
DOCKER     all  --  anywhere             anywhere

Chain DOCKER (1 references)
target     prot opt source               destination
ACCEPT     tcp  --  anywhere             172.17.0.2           tcp dpt:1234

# docker-proxy program forwards traffic from host port 1234 to container port 1234
> pgrep -af proxy
30676 /usr/bin/docker-proxy -proto tcp -host-ip 0.0.0.0 -host-port 1234 \
  -container-ip 172.17.0.2 -container-port 1234
```

Bridge Mode Limitations

- The container IP is hidden and cannot be used for service discovery
- Host ports become a limiting resource
- Service discovery must have host ip and port
- Port forwarding has a performance cost
- Does not scale well
- Application must support non standard port numbers
- Large scale solutions involve load balancers + service discovery

Overlay

The overlay network feature Uses VXLAN to create a private network. It is part of Docker swarm mode. Each group of containers (Pod) has a dedicated network which is the Kubernetes network model. It does not require any underlay network modification, i.e. the network that the hosts are using. The docker swarm integration is very well done and many of the details are nicely abstracted away.

Benefits include:

- Applications can use standard ports
- Simplified service discovery can use DNS

Overlay Example - Create Swarm

```
manager> docker swarm init --advertise-addr 128.224.56.106
Swarm initialized: current node (tqxsn8ytpdq8ntd4sswl6qxjo) is now a manager.
```

To add a worker to this swarm, run the following command:

```
docker swarm join \
--token SWMTKN-1-0f49cat8w4xm29qndjza1u294i2 128.224.56.106:2377

worker1> docker swarm join ...
This node joined a swarm as a worker.

manager> docker node ls
ID                            HOSTNAME         STATUS  AVAILABILITY  MANAGER STATUS
qshqkznzaty8ggbyiodzb9jy9     worker2          Ready   Active
r0fj6inhs1tsin07mdsxoaiam     worker1          Ready   Active
tqxsn8ytpdq8ntd4sswl6qxjo *   manager          Ready   Active        Leader
```

Swarm Network and Load Balancing

I found this great example from the Nginx example repository:

Docker Swarm Load Balancing

Docker Swarm has built in DNS, scheduling and load balancing! In the following example A is Service1, B is Service2 and is not externally accessible

Overlay Example - Create Service

```
> docker network create --driver overlay demo_net
> docker service create --name service1 --replicas=3 \
        --network demo_net -p 8111:80 service1
> docker service create --name service2 --replicas=3 --network demo_net service2
```

Containers spread across three machines

```
> docker service ps service1
5fl2xpzbka28  service1.1  service1  worker1  Running
u3f4bd8q3p6d  service1.2  service1  worker2  Running
i85jdtgtinxr  service1.3  service1  manager  Running
> docker service ps service2
b5bzfdqw10y2  service2.1  service2  worker1  Running
k39m6utcq56o  service2.2  service2  worker2  Running
uaftc3ax0k17  service2.3  service2  manager  Running
```

Overlay Example - Load Balancing

Service 1 contacts Service 2 using internal DNS. Swarm uses Round Robin DNS lookup by default.

```
manager> curl -s http://worker1:8111/service1.php | grep address
service1 address: 10.255.0.9
service2 address: 10.0.0.5
manager> curl -s http://worker2:8111/service1.php | grep address
service1 address: 10.255.0.8
service2 address: 10.0.0.4
```

Overlay Limitations

- VXLAN MTU and UDP complications
- VXLAN adds latency (10-20%) and reduces throughput (50-75%)
- Debugging VXLAN problems difficult
- Docker swarm hides all the setup and routing complexity
- Some network vendors provide VXLAN integration

Macvlan

- Linux Networking driver feature
- Low performance overhead
- MAC and IP per container, similar to VM
- MacVlan does not use VLANs!
- Recently moved from docker experimental

Macvlan Example

```
host1> docker network create --driver macvlan --subnet 128.224.56.0/24 \
    --gateway 128.224.56.1 -o parent=eth0 mv1
host2> docker network create --driver macvlan --subnet 128.224.56.0/24 \
    --gateway 128.224.56.1 -o parent=eth0 mv1
```

Choose unused IPs

```
host1> docker run -it --rm --net=mv1 --ip=128.224.56.119 alpine /bin/sh
host2> docker run -it --rm --net=mv1 --ip=128.224.56.120 alpine /bin/sh
/ # ping 128.224.56.119
PING 128.224.56.119 (128.224.56.119): 56 data bytes
64 bytes from 128.224.56.119: seq=0 ttl=64 time=0.782 ms
```

Imagine /16 subnet where each host has a /24 for container IPs

Macvlan Limitations

- Subnet and gateway must match host network
- Requires new kernels: 4.2+
- Requires IPAM and network cooperation
- Isolation requires VLANs and/or firewalls
- Limited to one broadcast domain
- Too many MACs can overflow NIC buffer
- Docker can allocate IPs in a given range
- IPVLan L2 mode very similar

IPVlan L3 Mode

- Linux Networking driver feature
- Low performance overhead
- Multicast and broadcast traffic silently dropped
- Mimics Internet architecture of aggregated L3 domains
- Scales well due to no broadcast domain
- Docker experimental as of 1.13

IPVlan Example

create network - requires dockerd run with –experimental

```
host1> docker network create --driver ipvlan --subnet 192.168.120.0/24 \
    -o parent=eth0 -o ipvlan_mode=l3 iv1
host2> docker network create --driver ipvlan --subnet 192.168.121.0/24 \
    -o parent=eth0 -o ipvlan_mode=l3 iv1
```

Setup routes: host1=128.224.56.106, host2=128.224.56.107

```
host1> ip route add 192.168.121.0/24 via 128.224.56.107
host2> ip route add 192.168.120.0/24 via 128.224.56.106
```

Create containers

```
host1> docker run -it --rm --net=iv1 --ip=192.168.120.10 alpine /bin/sh
host2> docker run -it --rm --net=iv1 --ip=192.168.121.10 alpine /bin/sh
/ # ping 192.168.120.10
PING 192.168.120.10 (192.168.120.10): 56 data bytes
64 bytes from 192.168.120.10: seq=0 ttl=64 time=0.408 ms
```

IPVLan Limitations

- Currently experimental
- Requires new kernels: 4.2+
- Isolation requires VLANs and/or iptables
- Manage routes using BGP with Calico, Cumulus, etc.
- Container networking becomes a routing problem, which is a well understood problem
- Policies using BPF on veth and Cillium

Conclusion

- Docker Multi-Host Networking is complicated!
- Performance and Scale dictate solution
- Balance between simplifying applications and infrastructure

Pages