Configuring Macvlan and Ipvlan Linux Networking

Posted MAR 28 2016 by BRENT SALISBURY in IN THE LAB

CONFIGURING MACVLAN AND IPVLAN LINUX NETWORKING

Macvlan and Ipvlan are both Linux type networking interfaces that are both supported by the Linux kernel. They are unique for a few different reasons. One thing that makes them both very attractive is they do not use bridges in their implementation and natively namespace aware. It solves some painful problems such as getting access to a gateway from a bridge that can alleviate a need for PAT/NAT and increased performance as a result of less overhead.

Traditionally we think of bridges as a common domain to attach interfaces to. In these two network types, the bridge is essentially replaced by a parent interface on the host in the default namespace. This would be a NIC on the host such as eth0 or a bonded pair of interfaces often named bond0. 802.1q trunk links are still supported just as they would be if the NIC was plumbed into a bridge.

> Update: Macvlan is now in production as of Docker v1.12. The docs for it can be found at: Macvlan Driver Docs
>
> To install the latest RC you can do the following:
>
> # Download Docker v1.12 RC2 Linux binary
> $ wget https://test.docker.com/builds/Linux/x86_64/docker-1.12.0-rc2.tgz
> $ tar xvzf docker-1.12.0-rc2.tgz
> $ cd docker
> $ cp docker* /usr/bin/
>
> # Start the daemon
> $ docker -d
>
> # Create a network (replace the subnet to match your eth interface)
> $ docker network create -d macvlan \
> –subnet=192.168.1.0/24 \
> –gateway=192.168.1.1 \
> -o parent=eth0 mcv1
>
> # Start a container on the new network
> $ docker run –net=mcv1 -it –rm alpine /bin/sh

We have recently added support for Ipvlan to the Docker Libnetwork project in experimental mode for the v1.11 release Ipvlan Network Drivers along with a slide deck the demonstrates the driver.

Personally, when I look at a new technology it helps to manually set something up to get a feel for whats happening under the covers. My buddy Scott Lowe recently did a great presentation at DevOps for Networking Forum on Linux Networking Types that covers these and more, so I recommend checking that out also.

For Diagrams of all of these scenarios see the diagrams we drew in the driver Readme link – Docker Network Experimental Macvlan and Ipvlan Driver Docs →

All of these use cases are done behind the scenes in the Docker drivers which we will write on much more in the future, but understanding the complexity that is being taken care of under the hood is a helpful with new technologies. All of the examples can be pasted directly into a Linux terminal and they will work. If you are on an old Linux distribution such as 14.10Ubuntu the iproute2 package is old and doesn't have Ipvlan in it. Either upgrade iproute2 or the distro. Quick example of iproute2 upgrade in this Gist.

MACVLAN BRIDGE MODE LINUX NETWORKING

First up is Macvlan Bridge mode. Macvlan will forward L2 broadcasts and multicast into the namespace. It also uses a unique MAC per Docker container. Note: if using VirtualBox it can be problematic getting multiple MACs off the Windows or OS X host. I recommend using NAT mode on the VirtualBox interface along with promiscuous mode for the VM if using VBox. VMware Fusion works with no problems, just have the interfaces to the VM on the Fusion side be promiscuous.

```
# add the namespaces
ip netns add ns1
ip netns add ns2

# create the macvlan link attaching it to the parent host eth0
```

```
ip link add mv1 link eth0 type macvlan mode bridge
ip link add mv2 link eth0 type macvlan mode bridge

# move the new interface mv1/mv2 to the new namespace
ip link set mv1 netns ns1
ip link set mv2 netns ns2

# bring the two interfaces up
ip netns exec ns1 ip link set dev mv1 up
ip netns exec ns2 ip link set dev mv2 up

# set ip addresses
ip netns exec ns1 ifconfig mv1 192.168.1.50/24 up
ip netns exec ns2 ifconfig mv2 192.168.1.60/24 up

# ping from one ns to another
ip netns exec ns1 ping 192.168.1.60

# cleanup
ip netns del ns1
ip netns del ns2
```

The macvlan/ipvlan Docker drivers will setup the VLAN tagging for the user instead of the user having to deal with making the configuration persistent with clunky config files. We create and delete sub-interfaces as networks get added and deleted. It also recreates all 802.1q trunks when the host reboots and Docker engine starts again.

For completeness, here is an example Linux VLAN sub-interface using ip link command. This would be used create a subinterface tagged with VLAN ID 20 on eth0.

```
# create a new subinterface tied to dot1q vlan 20
ip link add link eth1 name eth0.20 type vlan id 20

# assign an IP addr
ip a add 192.168.20.1/24 dev eth1.20

# enable the new sub-interface
ip link set eth0.20 up
```

All of the Macvlan and Vlan interface types are processed for you using the Macvlan Docker driver we developed with the following `docker network create` command. You can also create the ip links manually and pass them in as `-o parent=Foo0`.

```
docker network create -d macvlan \
    --subnet=192.168.215.0/24 \
    --gateway=192.168.215.1 \
    -o parent=eth0.20 macvlan215

# Test 192.168.215.0/24 connectivity
docker run --net=macvlan215 --ip=192.168.215.10 -itd alpine /bin/sh
docker run --net=macvlan215 --ip=192.168.215.9 -it --rm alpine ping -c 2 192.168.215.10
```

IPVLAN L2 MODE LINUX NETWORKING

Ipvlan L2 mode is virtually identical to Macvlan Bridge mode but instead has a single MAC address per container. There are upsides and downsides to this. Net is having unique mac-addresses hit a ceiling on most NICs of 500-1,000. While that would never be an issue in the VM world, the efficiencies you gain from containers 5x-10x+ and the resulting increase in density, creates a multiplier of more apps (and IP addresses) per server then we have ever seen before. That density is as good a reason as any to get a handle on container networking in your organization sooner rather then later.

NOTE: There is also a greater kernel requirement. While 3.19 supports Ipvlan, I would recommend >= v4.2. We also set the minimum kernel requirement in the Docker driver to v4.2 due to early bugs.

```
# add the namespaces
ip netns add ns3
ip netns add ns4
```

```
# create the macvlan link attaching it to the parent host eth0
ip link add ipv1 link eth0 type ipvlan mode l2
ip link add ipv2 link eth0 type ipvlan mode l2

# move the new interface ipv1/ipv2 to the new namespace
ip link set ipv1 netns ns3
ip link set ipv2 netns ns4

# bring the two interfaces up
ip netns exec ns3 ip link set dev ipv1 up
ip netns exec ns4 ip link set dev ipv2 up

# set ip addresses
ip netns exec ns3 ifconfig ipv1 192.168.1.10/24 up
ip netns exec ns4 ifconfig ipv2 192.168.1.20/24 up

# ping from one ns to another
ip netns exec ns3 ping 192.168.1.20

# cleanup
ip netns del ns3
ip netns del ns4
```

CONFIGURING IPVLAN L3 MODE LINUX NETWORKING

For anyone that has been in networking for a while, Ipvlan L3 mode will punish your senses of what can and can't be done. That is likely a good thing. First and foremost, there is no broadcast or multicast traffic allowed into the namespace. This means a next hop default gateway has little meaning. The default gateway is simply the namespace interface. If you are running a provider network, this presents excellent potentials for securing tenant traffic. Any network attached to the same parent interface can reach any other network attached to the same parent interface. This tends to remind me of a VRF like construct as it is a collection of networks in a namespace routing table. The potential for scale brings a lot of promise to the future of networking.

```
# add the namespaces
ip netns add ns5
ip netns add ns6

# create the macvlan link attaching it to the parent host eth0
ip link add ipv1 link eth0 type ipvlan mode l3
ip link add ipv2 link eth0 type ipvlan mode l3

# move the new interface ipv1/ipv2 to the new namespace
ip link set ipv1 netns ns5
ip link set ipv2 netns ns6

# bring the two interfaces up
ip netns exec ns5 ip link set dev ipv1 up
ip netns exec ns6 ip link set dev ipv2 up

# set ip addresses
ip netns exec ns5 ifconfig ipv1 10.1.100.10/24 up
ip netns exec ns6 ifconfig ipv2 192.168.100.10/24 up

# add default routes
ip netns exec ns5 ip route add default dev ipv1
ip netns exec ns6 ip route add default dev ipv2
# view the namespace default gateway (interface not next hop IP)
ip netns exec ns5 ip route
ip netns exec ns6 ip route

# ping from one ns to another
ip netns exec ns5 ping 192.168.100.10

# cleanup
ip netns del ns5
ip netns del ns6
```

Another option for Ipvlan L3 testing is if you are on your home network for example example, you can add a static route into the home router pointing the IP prefix "10.1.100.0/24" to the eth0 address on the host. That will enable the container to ping out to the Internet. Here is an example adding a route on a tp-link router. That could also be added to a ToR (top of rack switch). At the day job, we are working on partner integrations that will make plumbing those routes easy into a network fabric. There is a long list of examples in the script I used for debugging during dev here that highlights some of the killer default IPAM options that the Libnetwork team have added.

On the topic of L3, when people say L2 is bad and L3 is the future. There is a conversation that needs to happen with ops if you subscribe to that (which I tend to agree w/ for scale/stability). There is not a whole lot of difference between running a gateway protocol on a host that peers to a ToR vs. having integration between a ToR and a host's orchestration that will dynamically plumb routes. There is no other way to avoid flooding broadcasts and multicast to the application container running on the edge otherwise (and staying in the realm of reality).

STAY TUNED

These are great Linux networking options that have a lot of potential in both Enterprise and Provider data centers. Take a look at the drivers and readme to learn more about Macvlan and Ipvlan and how to get started using the Docker drivers to do some awesome. Also start thinking about how you want to distribute routes or even not by having static address endpoints and go stateless with some smart v4/v6 IP address planning or fancy bit-shifting.

Simplicity is ultimately the key. There are lots of L3 mode options on the horizon w/Ipvlan to distribute state coming up whether open source, vendor supported or hardware integrations, all of which have good scale, performance and isolation. In my experience, learning how the plumbing under the hood works is the first step in determining how to best design your DC/LAN/WAN. Hard not to see a converged and automated edge becoming the norm in most scenarios. The world changed over the past few years, its not a question of if but when. Step #1, don't make manual changes in your data center and figure out how to self-provision VLANs and routes without destroying your network in the process. Thats the value NetOps have over anyone else. The art of networking isn't learned in months, but years. Don't forget that the next time someone wants to hand wave away the networking details in your infra.

Macvlan is now in production as of Docker v1.12. The docs for it can be found at: Docker Macvlan Production Driver Docs →

Docker Network Experimental Ipvlan Driver Docs →

GoBGP – one of my favorite networking projects (blog shortly on it) →

We are currently looking for feedback on the Ipvlan drivers → Any and all feedback is welcome, even if simply saying the use case is valid in your environment is much appreciated. Thanks to @kelseyhightower for adding very useful feedback.

Time to Ditch the Definition of SDN GoBGP – A Control Plane Evolving Software Networking

TAGS: *Docker, Ipvlan, Libnetwork, Linux Networking, Macvlan* **CATEGORIES:** *In the Lab*

Brent Salisbury - I have over 15 years of experience wearing various hats from, network engineer, architect, devops and software engineer. I currently have the pleasure of working at the company that develops my favorite software I have ever used, Docker. My comments here are my personal thoughts and opinions. More at Brent's Bio
View all posts by Brent Salisbury →