Appendix A: Advanced Networking in Linux

In [linux], we discussed some basic Linux networking concepts. In this appendix, we'll use the building blocks from [linux] as a basis for discussing a few advanced Linux networking concepts and configurations.

The topics we'll cover in this appendix include:

- Using macvlan interfaces
- Networking virtual machines (VMs)
- Working with network namespaces
- Networking Linux containers
- Using Open vSwitch (OVS)

Many of these topics could be books on their own! Thus, our focus in discussing these topics won't be to provide comprehensive, in-depth coverage; instead, we'll focus on providing enough information for you to understand where these topics fit into the overall networking picture as well as the basics of how to install, configure, or manage these networking configurations.

We'll start with using macvlan interfaces.

Using macvlan Interfaces

The *macvlan interface* is sort of like the reverse of a VLAN interface, which we discussed in [linux]. VLAN interfaces allow a single physical interface to communicate in multiple VLANs (broadcast domains) simultaneously; you can think of this as a "many (networks) to one (physical interface)" arrangement. Contrast that to macvlan interfaces, which allow you to create multiple logical interfaces on a single broadcast domain—a "one (network) to many (logical interfaces)" arrangement. Each macvlan logical interface will have its own Media Access Control (MAC) address, and will only be able to see traffic destined for its MAC address. (One macvlan interface can't snoop on another macvlan interface's traffic, in other words.)

This may sound a bit esoteric, but there are at least a couple of use cases where this functionality can come in handy—let's explore those first.

Use Cases for macvlan Interfaces

Currently, macvlan interfaces have a couple use cases:

- If you're consolidating hosts and want to preserve the MAC address and IP address of hosts being retired, you can re-create those interfaces as macvlan interfaces on the new hosts. This will allow services to continue without any changes, even though the services are now running on a different host.
- You may also wish to use macvlan interfaces instead of a traditional Linux bridge in cases where you don't need the full functionality of the Linux bridge. We'll look at a couple examples of this, one in the next section and one later in the appendix.

Armed with this context of how macvlan interfaces could be used, we can dig into some of the technical details of working with macvlan interfaces.

Creating, Configuring, and Deleting macvlan Interfaces

To create a macvlan interface, you'll once again turn to the `ip` command—specifically, the `ip link` command. The generic syntax for the command to add a macvlan interface is **ip link add link** *parent-device macvlan-device* **type macvlan**.

Breaking that command down a bit:

- The *parent-device* is the physical interface with which the new macvlan interface should be associated. You may also see this referred to as the *lower device*.
- The *macvlan-device* is the name to be given to the new macvlan logical interface. Unlike with VLAN interfaces, there is no established naming convention.

So, let's say you wanted to create a macvlan interface on a AlmaLinux system (a RHEL/CentOS clone), and the new logical interface should be linked to the physical interface named ens33. The command would look like this:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip link add link ens33 macvlan0 type macvlan</strong>
[almalinux@alma9 ~]$
</pre>
```

If you wanted to create the macvlan interface with a specific MAC address (the previous command uses an auto-generated MAC address), then insert **address** *desired-MAC-address* between the macvlan device name and the `type macvlan` statement.

Once you've created the interface, you can verify that the interface was created using `ip link list`, and the `-d` parameter—which exposed additional information about VLAN interfaces—will expose additional information about macvlan interfaces:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip -d link list macvlan0</strong>
6: macvlan0@ens33: &lt;BROADCAST,MULTICAST&gt; mtu 1500 qdisc noop state DOWN mode DEFAULT
    link/ether b6:73:dc:60:a3:10 brd ff:ff:ff:ff:ff:ff promiscuity 0
    macvlan  mode vepa
</pre>
```

Note the `macvlan mode vepa` on the last line; this indicates the current mode of the macvlan device. A mode of vepa indicates that the Linux host expects the upstream switch to support 802.1Qbg, which—as of this writing—was fairly limited. Other modes are available; in addition to vepa, you can use bridge, private, and passthru. The bridge mode is *probably* what you'll want in most cases, and you can set the mode either when the interface is created or later.

To set the mode when the interface is created:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip link add link ens33 macvlan0 type macvlan mode
bridge</strong>
```

```
[almalinux@alma9 ~]$
</pre>
```

Or, if you need to set the mode after the interface has been created, use `ip link set`:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip link set macvlan0 type macvlan mode bridge</strong>
[almalinux@alma9 ~]$ <strong>ip -d link list macvlan0</strong>
6: macvlan0@ens33: &lt;BROADCAST,MULTICAST&gt; mtu 1500 qdisc noop state DOWN mode DEFAULT
    link/ether b6:73:dc:60:a3:10 brd ff:ff:ff:ff:ff:ff promiscuity 0
    macvlan  mode bridge
[almalinux@alma9 ~]$
</pre>
```

As with almost all other kinds of interfaces, you'll still need to enable the interface (set the state to up) and assign an IP address for the interface to be fully functional:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip link set macvlan0 up</strong>
[almalinux@alma9 ~]$ <strong>ip addr add 192.168.100.112/24 dev macvlan0</strong>
[almalinux@alma9 ~]$
</pre>
```

To delete a macvlan interface, first disable it with `ip link set`, then delete it with `ip link delete`:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip link set macvlan0 down</strong>
[almalinux@alma9 ~]$ <strong>ip link delete macvlan0</strong>
[almalinux@alma9 ~]$
</pre>
```

So how does one make macvlan interface configurations persistent? By default, RHEL/Fedora/CentOS systems (as of the time of writing) did not have a means whereby you could use a per-interface configuration file in */etc/sysconfig/network-scripts* to create persistent macvlan interface configurations. There are workarounds for this; for example, we found at least one GitHub repository that has scripts to make this possible.

On Debian/Ubuntu systems, there is a workaround that leverages the `pre-up` functionality in network configuration stanzas to run other commands before bringing up the network interface. This configuration, for example, would create a persistent macvlan interface associated with the eth2 physical interface:

```
auto macvlan0
iface macvlan0 inet static
  address 192.168.100.110/24
  pre-up ip link add link eth2 macvlan0 type macvlan
```

Networking Virtual Machines

Providing networking for VMs running on a Linux-based hypervisor is a topic that could be a book unto itself, but in this section we're going to attempt to tackle a couple of high-level configurations that should cover the majority of the implementations you're likely to encounter in the real world. To help keep the amount of material we need to cover manageable, we'll limit our discussion in this section to using the KVM hypervisor (as opposed to Xen or another Linux-based solution) and generally only with the Libvirt virtualization API. There are, of course, other hypervisors, other tools, and other configurations; unfortunately, we can't cover all possible combinations here.

The two VM networking configurations we'll discuss are:

- Networking VMs using a Linux bridge
- Networking VMs using macvtap interfaces

Let's start by looking at using a Linux bridge.

Using a Bridge

Bridging VMs onto a physical network via one (or more) of the Linux host's physical interfaces is a very common use case for the Linux bridge. In fact, it was one of the examples we used in [linux] when we first introduced bridging in Linux. In this section, we'll take a slightly more detailed look at how this works.

When you are networking VMs using a bridge with KVM and Libvirt, several different components come into play:

- A Linux bridge (naturally)
- A Libvirt virtual network that tells Libvirt which Linux bridge to use
- A virtual network interface
- A KVM guest domain (the word *domain* is used to refer to a VM running on KVM)

Let's see how all these pieces fit together.

Generally, one of the first things you'd do is use Libvirt to create a virtual network by defining it via some XML. (If you're not familiar with XML, no problem; refer to [dataformats].) A virtual network is an abstraction used by Libvirt to refer to a specific underlying networking configuration. The underlying network configuration could be a bridge (as in this case), or it could be some other configuration (as we'll see in the next section).

Libvirt uses XML for the definitions of its abstractions, including virtual networks. The following XML code would create a virtual network named network-br0 that references a Linux bridge named br0. Note that it's up to the system administrator to create br0 and associate a physical interface with the bridge, using the procedures and commands outlined in [linux].

```
<network>
  <name>network-br0</name>
  <forward mode="bridge"/>
  <bridge name="br0"/>
</network>
```

To tell a KVM domain to use this virtual network, you'd configure its domain XML to look something like this (we're only showing you the networking-relevant portion of the domain's XML definition, and not all possible options are included):

```
<interface type="network">
  <source network="network-br0"/>
</interface>
```

In this case, we're telling Libvirt (via this XML definition for a guest domain) to reference the Libvirt network named network-br0. This Libvirt network, in turn, references the Linux bridge named br0. The advantage of using the virtual network abstraction is that we could switch the underlying network bridge from br0 to br1 by simply modifying the virtual network definition. We wouldn't have to modify any of the VMs because they reference the virtual network.

With this configuration in place, when the guest domain is started KVM and Libvirt will automatically create a virtual network interface (a TAP device) and attach it to the bridge specified by the Libvirt virtual network definition (in this case, br0). The guest domain will have its own network interface, which will be associated by the hypervisor with the TAP device. This creates a "chain" of connectivity: the guest's eth0 is connected to the TAP device, which is connected to the bridge, which is connected to the physical interface and the network beyond.

Libvirt automates almost all of this for you, which can make it a bit more difficult to observe it in action. It's possible, though, to manually set all this up so that you can see how the pieces fit together. The next few paragraphs will walk you through the steps required to manually bridge a VM onto a network. We don't recommend this for any sort of production use, but it can be useful as a learning exercise to better understand what's happening "behind the scenes" with KVM and Libvirt.

First, you'll want to create the Linux bridge and attach a physical interface to the bridge. Assuming that eth1 is the interface you want to attach to the bridge, you'd run commands like these:

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ip link add name br0 type bridge</strong>
ubuntu@ubuntu2004:~$ <strong>ip link set br0 up</strong>
ubuntu@ubuntu2004:~$ <strong>ip link set eth1 master br0</strong>
ubuntu@ubuntu2004:~$ <strong>ip link set eth1 up</strong>
ubuntu@ubuntu2004:~$
</pre>
```

NOTE

Note that whether you are manually attaching VMs to a bridge or using Libvirt, you would still have to use the various `ip` commands to create the Linux bridge and attach one (or more) interfaces. Note that the interfaces you attach to the bridge could be VLAN interfaces!

Next, you'd want to create the TAP device using a new command we haven't shown you yet: the `ip` `tuntap` command. The generic form of the command to add a TAP device is **ip tuntap add** *dev-name* **mode tap**. If we wanted to use the name tap0 for the TAP device to which we'll connect our VM, we run these commands:

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ip tuntap add tap0 mode tap</strong>
ubuntu@ubuntu2004:~$ <strong>ip link set tap0 up</strong>
ubuntu@ubuntu2004:~$ <strong>ip -d link list tap0</strong>
5: tap0: &lt;NO-CARRIER,BROADCAST,MULTICAST,UP&gt; mtu 1500 qdisc pfifo_fast state DOWN
    mode DEFAULT group default qlen 500
    link/ether 7e:28:d5:99:ca:ab brd ff:ff:ff:ff:ff:ff promiscuity 0
    tun
ubuntu@ubuntu2004:~$
</pre>
```

The first command creates the TAP device, the second command enables the link, and the third command verifies the status of the device. The output of the third command tells you the interface is enabled but not connected to anything (note the NO-CARRIER in the output).

Next, add the TAP device to the existing bridge, then verify using the `bridge link` command:

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ip link set tap0 master br0</strong>
ubuntu@ubuntu2004:~$ <strong>bridge link list</strong>
3: eth1 state UP : &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 1500 master br0 state
forwarding priority 32 cost 4
5: tap0 state DOWN : &lt;NO-CARRIER,BROADCAST,MULTICAST,UP&gt; mtu 1500 master br0 state
disabled priority 32 cost 100
ubuntu@ubuntu2004:~$
</pre>
```

The final step is to launch a virtual machine and attach it to the TAP device. We won't go into any great detail on the command used here, if for no other reason than we think it's unlikely you'll need it in real-world usage (you're far more likely to use the `virsh` command that comes with Libvirt). Note that the command is line-wrapped with backslashes here to make it more readable.

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>qemu-system-x86_64 -enable-kvm -hda cirros-01.qcow2 \
-net nic -net tap,ifname=tap0,script=no,downscript=no -vnc :1 &#x26;</strong>
[1] 866
ubuntu@ubuntu2004:~$
</pre>
```

This will boot a KVM domain in the background. If you now run `ip -d link list tap0`, you'll see that the TAP device is active (note that `bridge link list` would also show you the TAP device is up and active):

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ip -d link list tap0</strong>
5: tap0: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 1500 qdisc pfifo_fast master br0
    state UP mode DEFAULT group default qlen 500
    link/ether 7e:28:d5:99:ca:ab brd ff:ff:ff:ff:ff:ff promiscuity 1
    tun
    bridge_slave
ubuntu@ubuntu2004:~$
</pre>
```

If you have a DHCP server running on the network segment to which the KVM host's eth1 is connected, then your KVM guest domain should obtain an IP address and be reachable from other systems on the same subnet.

Again, let us reiterate that you *don't* have to perform all the manual steps we outlined here to use bridged networking with KVM and Libvirt. We included the manual steps here to help you better understand all the various pieces that are involved. KVM and Libvirt automate the majority of these steps.

Also, now that we've covered VLAN interfaces we can point out that you can also use VLAN interfaces in a bridge. This might be one way of bridging different VMs on a single Linux hypervisor onto different VLANs—create a bridge for each VLAN, add a VLAN interface to each bridge, and then attach VMs to that bridge.

While using a bridge is one (very common) way to provide networking for VMs, it's by far not the only way. Later in this appendix in Using Open vSwitch, we'll talk about how to use Open vSwitch (OVS) to provide networking for VMs. First, though, let's take a look at another way of providing network connectivity to VMs: macvtap interfaces.

Using macvtap Interfaces

In Using macvlan Interfaces we showed you how to use macvlan interfaces to configure a Linux system with multiple network identities on a single physical interface. A close relative (it uses the same Linux kernel driver) of the macvlan interface is the macvtap interface, which allows us to use these multiple identities to provide network connectivity for VMs.

To use macvtap interfaces with KVM and Libvirt, you'd again first start with defining a Libvirt virtual network that references macvtap interfaces. This snippet of XML would allow you to define a virtual network named macvtap-net that leverages macvtap interfaces running in bridge mode and is associated with the eth1 physical interface:

```
<network>
  <name>macvtap-net</name>
  <forward mode="bridge">
    <interface dev="eth1"/>
  </forward>
</network>
```

Just as when we use a bridge with KVM and Libvirt, the domain XML configuration then needs to only reference the Libvirt network:

```
<interface type="network">
```

```
  <source network="macvtap-net"/>
</interface>
```

When you start/launch a VM using Libvirt, it will automatically create a macvtap interface on the associated physical interface. You can verify this by running `ip link list`; you should see a macvtap interface in the output.

One interesting side effect, if you will, of using macvtap interfaces is that the MAC address seen inside the guest domain will be the same as the MAC address used by the macvtap interface. For example, here's the output of `ip link list eth0` from within a guest domain when using a macvtap interface:

```
2: eth0: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast qlen 1000
    link/ether 52:54:00:9c:51:74 brd ff:ff:ff:ff:ff:ff
```

For comparison, here's the output of `ip link list macvtap0` on the host system, where macvtap0 is the macvtap interface created by Libvirt when the guest domain was launched:

```
5: macvtap0@eth1: <BROADCAST,MULTICAST,UP,LOWER_UP> mtu 1500 qdisc pfifo_fast state
    UNKNOWN mode DEFAULT group default qlen 500
    link/ether 52:54:00:9c:51:74 brd ff:ff:ff:ff:ff:ff
```

This direct correlation between the MAC address inside the guest and the MAC address outside the guest may simplify some troubleshooting and/or information gathering efforts.

We're going to discuss one other way of providing networking for VMs (using Open vSwitch), but before we do that we're going to take a slight detour into a couple other advanced Linux networking topics.

Working with Network Namespaces

The basics of network namespaces are covered in Chapter 4, "Cloud". This section extends that coverage by providing some additional information on working with network namespaces directly, not via a container runtime or container orchestrator.

Creating and Removing Network Namespaces

Creating a network namespace is really pretty straightforward. The tool of choice is again the `ip` command from the iproute2 package, this time using the netns set of subcommands.

To create a network namespace, the syntax for the command is **ip netns add** *namespace-name*. As an example, let's say that you wanted to create a namespace called blue:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip netns add blue</strong>
[almalinux@alma9 ~]$
</pre>
```

Note there’s no feedback for a successful command; to verify the namespace was added, you’ll need to use `ip netns list`:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip netns list</strong>
blue
[almalinux@alma9 ~]$
</pre>
```

Deleting network namespaces is equally straightforward:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip netns del blue</strong>
[almalinux@alma9 ~]$ <strong>ip netns list</strong>
[almalinux@alma9 ~]$
</pre>
```

The lack of output from the `ip netns list` command indicates there are no network namespaces other than the "default" namespace in which all networking objects normally reside.

While adding and deleting namespaces is (somewhat) interesting, the real value lies in actually *using* network namespaces. To do that, we’ll first need to look at how to assign interfaces to a particular namespace.

Placing Interfaces in a Network Namespace

By default, all of the networking-related objects and configurations belong to the "default" network namespace (also known as "netns 0"). Also by default, a newly created network namespace contains no network interfaces. Thus, a newly created network namespace has *no* network connectivity to anything: not to the default namespace, not to the outside world, not to *anything*. To fix that, you need to place an interface into the namespace.

To place an interface into a namespace, use the `ip link` command (obviously this command assumes that the blue namespace has already been created):

```
<pre data-type="programlisting">
debian@debian11:~$ <strong>ip link set ens3 netns blue</strong>
debian@debian11:~$
</pre>
```

As you can tell from this example, the general syntax to place an interface into a network namespace is **ip link set** *interface-name* **netns** *namespace-name*.

Once you place an interface into a namespace, it disappears from the default namespace. This makes sense, because an interface can exist in only a single namespace at any given time.

Although we’ve only shown you examples that assign a physical interface to a namespace, you’re not limited to physical interfaces. Suppose you wanted to assign a VLAN interface to a namespace:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip link set ens33.150 netns blue</strong>
```

```
[almalinux@alma9 ~]$
</pre>
```

Or suppose you want to assign a macvlan interface to a particular namespace:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip link set macvlan0 netns red</strong>
[almalinux@alma9 ~]$
</pre>
```

This gives you a great deal of flexibility in how you connect network namespaces to the outside world.

Regardless of the type of interface, the command to assign it to a namespace remains the same: **ip link set** *interface-name* **netns** *namespace-name*. And regardless of the type of interface, once it is assigned to a namespace it disappears from the default namespace. To work with any interface assigned to a non-default namespace, you need to run commands within the context of the namespace in which it resides. In other words, you're going to need to execute commands *inside* a particular namespace.

Executing Commands in a Network Namespace

To execute a command in the context of a specific network namespace, you'll need to use the `ip netns exec` command. The general syntax for this command is **ip netns exec** *namespace-name command*. Let's look at a few examples.

In the previous section, we used the `ip link set` command to assign the ens3 interface on a Debian 11 system into the blue namespace. If we now want to be able to see that interface, we'll combine `ip netns exec` (to execute a command inside a particular namespace) with `ip link list` (to show us the list of network interfaces), like this:

```
<pre data-type="programlisting">
debian@debian11:~$ <strong>ip netns exec blue ip link list</strong>
1: lo: &lt;LOOPBACK&gt; mtu 65536 qdisc noop state DOWN mode DEFAULT group default
    link/loopback 00:00:00:00:00:00 brd 00:00:00:00:00:00
3: ens3: &lt;BROADCAST,MULTICAST&gt; mtu 1500 qdisc noop state DOWN mode DEFAULT group
    default qlen 1000
    link/ether 00:0c:29:7d:38:9d brd ff:ff:ff:ff:ff:ff
debian@debian11:~$
</pre>
```

We can see from this output that the ens3 interface exists inside the blue namespace, but is currently disabled (note state DOWN in the output). To enable this interface:

```
<pre data-type="programlisting">
debian@debian11:~$ <strong>ip netns exec blue ip link set ens3 up</strong>
debian@debian11:~$ <strong>ip netns exec blue ip link list ens3</strong>
3: ens3: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 1500 qdisc pfifo_fast state UP
    mode DEFAULT group default qlen 1000
    link/ether 00:0c:29:7d:38:9d brd ff:ff:ff:ff:ff:ff
debian@debian11:~$
```

```
</pre>
```

Now the interface is up, and we could assign an IP address and check the namespace's routing table:

```
<pre data-type="programlisting">
debian@debian11:~$ <strong>ip netns exec blue ip addr add 192.168.100.10/24 dev
ens3</strong>
debian@debian11:~$ <strong>ip netns exec blue ip route list</strong>
192.168.100.0/24 dev ens3  proto kernel  scope link  src 192.168.100.11
debian@debian11:~$
</pre>
```

To prove that the namespaces are separate—in other words, that the IP configuration within the blue namespace does not affect the default namespace—run the `ip route list` command in the default namespace as follows:

```
<pre data-type="programlisting">
debian@debian11:~$ <strong>ip route list</strong>
default via 192.168.70.2 dev ens5
192.168.70.0/24 dev ens5  proto kernel  scope link  src 192.168.70.242
debian@debian11:~$
</pre>
```

The IP configuration and associated route linked to eth1 no longer affect the default namespace, only the blue namespace where the interface is assigned. (We'll leave it as an exercise for the readers to check the routing table in the blue namespace.)

Now that we have an interface that is assigned to a namespace, is enabled, and has an IP address configured, we can test connectivity from that specific namespace to the outside world using `ip netns exec` and the ubiquitous `ping` command:

```
<pre data-type="programlisting">
debian@debian11:~$ <strong>ip netns exec blue ping -c 4 192.168.100.100</strong>
</pre>
```

Throughout all these examples we're showing, you may have noticed that we keep having to type `ip netns exec` in front of commands in order to execute them in a particular namespace. Here, you may find leveraging bash's alias functionality—the ability to create commands that reference other commands—to be extraordinarily helpful. For example, you could define the alias `nsblue` to execute commands inside the blue network namespace:

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>alias nsblue="ip netns exec blue"</strong>
ubuntu@ubuntu2004:~$
</pre>
```

With this alias defined, you can now just type `nsblue` instead of `ip netns exec blue` when you want to execute commands inside the blue network namespace.

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>nsblue ip link list</strong>
```

```
3: ens3: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 1500 qdisc pfifo_fast state UP
    mode DEFAULT group default qlen 1000
    link/ether 00:0c:29:7d:38:9d brd ff:ff:ff:ff:ff:ff
ubuntu@ubuntu2004:~$
</pre>
```

Although these examples show physical interfaces being assigned to a network namespace, remember that you can assign just about any type of interface—physical interfaces, VLAN interfaces, macvlan interfaces, or virtual Ethernet (veth) interfaces—to a network namespace. When you assign one of these types of interfaces to a network namespace, though, you're connecting that namespace to the outside world (a particular VLAN if you're using a VLAN interface, for example).

Connecting Network Namespaces with veth Pairs

As described in Chapter 4, connecting network namespaces with veth pairs is the foundation for container networking in Linux. When working with a container runtime or a container orchestrator, the mechanics of setting up network namespaces and veth pairs are handled without any real need for the user to get involved. This section digs a bit deeper into the specifics of how a user might manually create veth pairs and use them to connect network namespaces.

Let's take a quick look at how this works. First, you'll create the veth pair using the `ip` command. The syntax for the command is **ip link add** *veth-name* **type veth peer name** *veth-peer*. If you wanted to create a veth pair named veth0 and veth1, then the command would look like this:

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ip link add veth0 type veth peer name veth1</strong>
ubuntu@ubuntu2004:~$ <strong>ip -d link list veth0</strong>
5: veth0: &lt;BROADCAST,MULTICAST&gt; mtu 1500 qdisc noop state DOWN mode DEFAULT group
    default qlen 1000
    link/ether f6:67:c0:f8:75:7d brd ff:ff:ff:ff:ff:ff promiscuity 0
    veth
ubuntu@ubuntu2004:~$
</pre>
```

Any traffic that enters either member of the veth pair will exit the other member of the veth pair. So, if we place veth1 into a network namespace, then traffic that enters veth1 in whatever namespace we place it will exit veth0 in the default namespace, thus connecting the two namespaces together.

In the following set of commands, we'll create a network namespace called green, then place veth1 into that namespace. We'll then use `ip netns exec` to configure veth1, and then test connectivity between the two namespaces.

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ip netns add green</strong>
ubuntu@ubuntu2004:~$ <strong>ip link set veth1 netns green</strong>
ubuntu@ubuntu2004:~$ <strong>ip netns exec green ip addr add 10.0.3.1/24 dev
veth1</strong>
ubuntu@ubuntu2004:~$ <strong>ip netns exec green ip link set veth1 up</strong>
ubuntu@ubuntu2004:~$ <strong>ip addr add 10.0.3.2/24 dev veth0</strong>
```

```
ubuntu@ubuntu2004:~$ <strong>ip link set veth0 up</strong>
ubuntu@ubuntu2004:~$ <strong>ping -c 4 10.0.3.1</strong>
PING 10.0.3.1 (10.0.3.1) 56(84) bytes of data.
64 bytes from 10.0.3.1: icmp_seq=1 ttl=64 time=0.046 ms
64 bytes from 10.0.3.1: icmp_seq=2 ttl=64 time=0.078 ms
64 bytes from 10.0.3.1: icmp_seq=3 ttl=64 time=0.066 ms
64 bytes from 10.0.3.1: icmp_seq=4 ttl=64 time=0.077 ms

--- 10.0.3.1 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3002ms
rtt min/avg/max/mdev = 0.046/0.066/0.078/0.016 ms
ubuntu@ubuntu2004:~$
</pre>
```

So what did we just do here? We already had the veth pair, so we created a namespace named green and placed veth1 into that namespace. We then assigned an IP address to veth1, and enabled the interface. Then, *so that the default namespace had a route to the destination*, we added an IP address to veth0. We then pinged between the two network namespaces.

What if you wanted to connect a network namespace to the outside world using veth pairs? No problem—create the veth pair, place one veth interface in the network namespace, and then put the other veth interface into a bridge with a physical interface. Your network namespace is now bridged to the outside world. (We'll leave it to you to do this as a learning exercise.)

Naturally, we could create more complex topologies, but this gives you an idea of what's possible using veth pairs to connect network namespaces.

In the next section, we'll take a look at a practical application of network namespaces: Linux containers.

Networking Linux Containers with LXC

Docker is not the only container game in town. An older (and some might say more mature) approach is known as LXC (which stands for *L*inu*X* *C*ontainers).

Comparing and Contrasting Docker and LXC

Docker and LXC leverage the exact same kernel features (namespaces for isolation and cgroups for resource accounting and limiting); where they differ is in how they build their containers and how a user leverages their containers. Each approach has its advantages and disadvantages.

What LXC and Docker do share (in addition to their use of the same underlying Linux kernel constructs) are certain facets of how they do container networking (all these are default settings):

- Both LXC and Docker leverage veth pairs, placing one of the veth interfaces into a network namespace with the container and leaving the peer interface in the default network namespace.
- Both LXC and Docker leverage a Linux bridge to which the veth peer interface in the default namespace is attached. The default LXC bridge is named lxcbr0, whereas the default Docker bridge is named docker0.

- Both LXC and Docker use custom iptables rules to perform network address translation (NAT) for container connectivity.

We've discussed all these mechanisms in previous sections, so you should already be familiar with veth interfaces, placing veth interfaces into a network namespace, and using bridges to provide connectivity.

Although you can see that LXC and Docker share a fair number of similarities, there are also quite a few differences, especially in terms of how you configure network settings for each. Let's take a closer look at configuring container networking for both LXC and Docker.

Configuring LXC Networking

LXC stores networking configuration on a per-container basis. On Ubuntu systems (this path may vary from distribution to distribution), the file */var/lib/lxc/<container-name>/config* contains some critical LXC networking configuration options:

- The `lxc.network.type` option controls the networking type for the containers. The default is veth, which tells LXC to use veth pairs. You can also specify macvlan, which tells LXC to use macvlan interfaces. In the event of using macvlan interfaces, you can use the `lxc.network.macvlan.mode` to set the mode (private, vepa, bridge) of the macvlan interfaces. LXC also supports a value of vlan, which means containers will leverage a VLAN interface for connectivity.
- The `lxc.network.link` setting controls the bridge to which the LXC will be connected in the default namespace. By default, this value is lxcbr0. Leaving this value blank means that the container won't be connected to a bridge. Later in this appendix in Using Open vSwitch, we'll show you a use case for leaving this setting blank.
- The `lxc.network.veth.pair` option specifies the name of the veth pair that will sit outside the container namespace (the other member of the pair will be moved into the container's network namespace). This lets you control the naming convention used for the veth peer that remains in the default network namespace.
- The `lxc.network.ipv4`, `lxc.network.ipv4.gateway`, `lxc.network.ipv6`, and `lxc.network.ipv6.gateway` settings control IPv4 and IPv6 configuration for the container, respectively.

In short, LXC provides pretty extensive control over how networking is provided for containers.

NOTE

What about ipvlan interfaces?

In this section we've discussed a few different types of logical interfaces: VLAN interfaces, macvlan interfaces, and veth pairs, for example. We haven't discussed *ipvlan interfaces*, which are like macvlan interfaces but are differentiated at Layer 3 using IP addresses instead of at Layer 2 using MAC addresses. The support for ipvlan interfaces is still quite new, though, and ipvlan interfaces really only have a use case in container networking.

Using Open vSwitch

Open vSwitch (OVS) is an open source, production-quality multi-layer virtual switch designed to run within a hypervisor (although, as we'll see later, OVS has lots of applications besides just using it with a hypervisor). OVS was designed with network automation in mind, built to support programmatic control while still supporting a wide range of management protocols and standards. OVS was also designed from the ground up to support OpenFlow, the seminal SDN protocol, and is considered by many to be *the* definitive reference OpenFlow implementation. OVS is also widely supported: both the Xen and KVM hypervisors support OVS, and at the time of this writing a port of OVS to Hyper-V was nearly complete (it will likely be complete by the time this book makes it to print). Numerous management and orchestration systems, including OpenStack, have support for OVS.

Given its prominent role in the SDN and network automation spaces, it's fully expected that we should provide coverage of OVS in a book on network automation. However, because of the broad swath of features that OVS supports, we'll have to constrain our discussion. As a result, we'll focus on three core areas:

- Installing OVS (discussing OVS on Linux only)
- Configuring OVS
- Connecting workloads to OVS

Let's start at the beginning, and that's installing OVS.

Installing OVS

Due to OVS's architecture—comprising both a userspace daemon as well as a kernel module—the procedure for installing OVS varies depending on your Linux distribution and which version of the OVS kernel module you want to use.

Since version 3.3, the upstream Linux kernel has shipped with an OVS module. To use the upstream kernel module, no further action is required; you need only to install the userspace components. If, on the other hand, you prefer to use the kernel module from the OVS tree (which may be newer than the upstream module and therefore support more features), then you'll need to install and compile a kernel module for the currently running kernel.

NOTE If you'd like to verify whether your Linux kernel supports the upstream OVS kernel module, just run *modprobe openvswitch* (you may need to use `sudo` if you don't have superuser privileges). If the command reports an error, your kernel doesn't have the OVS upstream module, and you'll need to install a kernel module. Keep in mind, though, that the upstream OVS module has been in the Linux kernel since version 3.3, so virtually *all* modern distributions will have the upstream OVS module available in the kernel.

Starting with Debian 8.x and Ubuntu 14.04, installation packages for both the userspace components and the kernel module are available in the primary repositories. Installation, therefore, is just a matter of using `apt-get install`:

- To use the upstream kernel module, just install the userspace packages. The names of the userspace components are openvswitch-common and openvswitch-switch.
- To use the kernel module that ships with OVS, also install the openvswitch-datapath-dkms package (and the necessary prerequisites, dkms, make, and libc6-dev).

NOTE We stated that starting with Debian 8.x, there are packages for OVS in the primary repositories. However, you should be aware that, depending on your installation method, the primary repositories may not be enabled. Check your repository configuration in */etc/apt/sources.list* if you are unsure (you can use the `cat` command to view the configuration, and edit it to enable the primary repositories if necessary).

On RHEL/CentOS/Fedora, the story is—as of this writing—a bit more complicated. RHEL 7.x and CentOS 7.x did not ship with a repository enabled that contains OVS. The same is true for RHEL 8.x and CentOS 8.x, as well as Rocky Linux 8.x and AlmaLinux 8.x In order to install OVS, you either have to compile from source, or add a repository that contains OVS packages. One such repository is the OpenStack repository from the CentOS Cloud Special Interest Group (SIG). You can enable this repository by running `yum install centos-release-openstack`; when that command completes, verify the repository has been added using `yum repolist`. Fedora, however, ships with OVS packages available in the default Fedora repositories; no additional repositories need to be added or enabled.

To install OVS on RHEL/CentOS/Fedora once you have an available package, it's just a matter of running `yum install` (or `dnf install`) to install the openvswitch package.

Note that as of this writing RHEL/CentOS/Fedora don't offer a package to install the kernel module from the OVS tree; if you want that kernel module, you'll have to manually compile it and install it. Given that manually compiling and installing a kernel module is a fairly in-depth topic, it's not something we'll discuss here. There are, however, a number of guides available online from various sources.

Once you have OVS installed, we can move on to our next section: configuring OVS.

Configuring OVS

OVS can be configured in a couple of different ways: you can use the OVS-specific command-line tools, or you can leverage OVS integration into the Linux network configuration scripts (*/etc/network/interfaces* on Debian/Linux, */etc/sysconfig/network-scripts* on RHEL/CentOS/Fedora). In this section, we'll focus primarily on the use of the OVS-specific command-line tools. The reason for this is that OVS *doesn't* follow the general Linux convention of needing to edit configuration files in order for a configuration to be persistent.

That's right—changes you make to the OVS configuration using the OVS command-line tools are persistent. OVS maintains its own configuration database, and the OVS command-line tools manipulate that database. When a system is restarted, OVS will reread its configuration from the configuration database; thus, every change you make to OVS using the OVS command-line tool *is* a persistent change. This is a key difference with OVS versus a lot of the other network configurations we discussed in [linux] and in this appendix.

The primary tool you will use to configure OVS is `ovs-vsctl`. Like the `ip` commands we discussed both here and in [linux], the `ovs-vsctl` command has a number of subcommands for various purposes:

- The `show` subcommand simply prints an overview of the configuration database's contents (i.e., prints an overview of OVS's configuration).

- The `add-br` command adds an OVS bridge to the OVS configuration. Any OVS bridge is conceptually and functionally similar to the Linux bridge, but with a vastly expanded set of capabilities.
- The `del-br` command deletes an OVS bridge.
- The `add-port` command adds a port to an OVS bridge. Ports can be physical interfaces (like eth1 or ens33) or logical network interfaces (like a VLAN interface or a veth interface).
- Similarly, the `del-port` command removes a port from an OVS bridge.

There are more commands, but these comprise the bulk of the functionality you'll need to get started with OVS. Let's look at some examples.

Assuming you have OVS installed and running, let's start by creating an OVS bridge. First, we'll show the current configuration (to show that it is empty—that OVS is essentially unconfigured), and then we'll add a bridge and show the configuration again. The syntax for adding a bridge to OVS is **ovs-vsctl add-br** *bridge-name*. Here's the command in action:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ovs-vsctl show</strong>
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    ovs_version: "2.4.0"
[almalinux@alma9 ~]$ <strong>ovs-vsctl add-br br0</strong>
[almalinux@alma9 ~]$ <strong>ovs-vsctl show</strong>
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    Bridge "br0"
        Port "br0"
            Interface "br0"
                type: internal
    ovs_version: "2.4.0"
[almalinux@alma9 ~]$
</pre>
```

You now have an OVS bridge—but like a Linux bridge, it doesn't really *do* anything until you add some ports. Let's add the physical ens33 interface to this bridge:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ovs-vsctl add-port br0 ens33</strong>
</pre>
```

As you can see, the syntax for adding a port to a bridge is **ovs-vsctl add-port** *bridge-name port-name*. With one exception that we'll discuss later, the port you're adding to OVS needs to already exist and be recognized by Linux.

Running `ovs-vsctl show` now will show the physical port has been added:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ovs-vsctl show</strong>
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    Bridge "br0"
        Port "ens33"
            Interface "ens33"
        Port "br0"
            Interface "br0"
                type: internal
    ovs_version: "2.4.0"
</pre>
```

To delete a port or a bridge, you'd use the `del-port` or `del-br` commands, respectively:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ovs-vsctl del-port br0 ens33</strong>
[almalinux@alma9 ~]$ <strong>ovs-vsctl del-br br0</strong>
[almalinux@alma9 ~]$ <strong>ovs-vsctl show</strong>
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    ovs_version: "2.4.0"
[almalinux@alma9 ~]$
</pre>
```

In addition to the subcommands we've shown you so far, you may also find yourself needing to use the `set` subcommand to set properties or values. For example, to apply a VLAN tag to an OVS port, you'd use the command syntax **ovs-vsctl set port** *port-name tag*=*value*. Suppose you have a port named vnet0 that represents a VM (this is a scenario we'll discuss shortly in Using VMs with OVS), and you want that VM to be on VLAN 10. You'd use this command:

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ovs-vsctl set port vnet0 tag=10</strong>
ubuntu@ubuntu2004:~$ <strong>ovs-vsctl show</strong>
fe63a9ea-f72f-4aa2-b390-42ecbed6deef
    Bridge "br0"
        Port "vnet0"
            tag: 10
            Interface "vnet0"
        Port "br0"
            Interface "br0"
                type: internal
        Port "eth1"
            Interface "eth1"
    ovs_version: "2.0.2"
ubuntu@ubuntu2004:~$
</pre>
```

You may also find the `list` subcommand helpful, as it will list all the properties/values associated with a configuration object in the OVS configuration database. If you wanted to see all the configuration values for the vnet0 port, you'd run this command:

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ovs-vsctl list port vnet0</strong>
_uuid                : cc51fc7e-ce14-41c6-9ad6-7b3ae717afa9
bond_downdelay       : 0
bond_fake_iface      : false
bond_mode            : []
bond_updelay         : 0
```

```
external_ids        : {}
fake_bridge         : false
interfaces          : [74e6ede7-1a13-45c1-84d6-f66cbfc5a353]
lacp                : []
mac                 : []
name                : "vnet0"
other_config        : {}
qos                 : []
statistics          : {}
status              : {}
tag                 : 10
trunks              : []
vlan_mode           : []
ubuntu@ubuntu2004:~$
</pre>
```

There's obviously much, much more—like creating overlay networks with a protocol like VXLAN or Geneve, working with OpenFlow flows, or setting OVS to use an external controller—but the majority of what you'll do with OVS will involve adding and removing bridges, adding and removing ports, and setting properties on ports.

Let's turn our attention now to putting some of the commands we've shown you in the section to work as we look at connecting various types of workloads to OVS.

Connecting Workloads to OVS

Here we'll use the term *workloads* to refer to any sort of entity that needs network connectivity—this could be a network namespace, a container, a virtual machine (like a KVM guest domain), or the OVS host system itself.

The process for connecting workloads to OVS will vary based on a variety of factors, but it will generally look like this:

- For network namespaces and containers, you'll often use a veth pair to connect a network namespace to OVS.
- For KVM guest domains, attaching to OVS is typically handled via a TAP interface.
- For the host system, you can direct traffic through OVS by using an OVS internal port.

Let's take a look at each of these scenarios in a bit more detail. Refer back to previous sections if you need a refresher on any of the commands used.

Connecting network namespaces with OVS

Recall from our earlier discussion on network namespaces that one way to connect network namespaces is to use a veth pair. One of the veth interfaces is placed in a network namespace (using the `ip link set` command), and the peer interface remains in the primary namespace.

We can use veth pairs with OVS to connect network namespaces to OVS (and thus to any sort of network topology that OVS supports—a physical network or an overlay network). To do this, we'd use the same basic setup we described earlier to bridge a network namespace onto the network.

Assuming you have a network namespace named green, then you'd first create the veth pair, place one of the veth interfaces into the green namespace, and configure the interface in the green namespace:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip link add veth0 type veth peer name veth1</strong>
[almalinux@alma9 ~]$ <strong>ip link set veth1 netns green</strong>
[almalinux@alma9 ~]$ <strong>ip netns exec green ip addr add 192.168.100.12/24 dev
veth1</strong>
[almalinux@alma9 ~]$ <strong>ip netns exec green ip link set veth1 up</strong>
[almalinux@alma9 ~]$ <strong>ip link set veth0 up</strong>
</pre>
```

At this point, you have a veth pair (veth0 and veth1), and the veth1 interface has been assigned to the green interface and given an IP address. Both veth interfaces are also up (enabled), so that traffic will flow between them. Now, to connect the green network namespace to OVS, just add veth0 to an OVS bridge. Let's assume you already have an OVS bridge named br0, and that bridge also contains the ens33 physical interface:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ovs-vsctl add-port br0 veth0</strong>
[almalinux@alma9 ~]$ <strong>ovs-vsctl show</strong>
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    Bridge "br0"
        Port "veth0"
            Interface "veth0"
        Port "br0"
            Interface "br0"
                type: internal
        Port "ens33"
            Interface "ens33"
    ovs_version: "2.4.0"
[almalinux@alma9 ~]$
</pre>
```

You can see that we just used the `ovs-vsctl add-port` command, along with the name of the bridge (br0) and the name of the interface to add (veth0). The network namespace is now connected to OVS (in this particular configuration, we've just bridged the network namespace onto the network connected to the ens33 physical interface).

Naturally, once you have a network namespace connected to OVS, it can then take advantage of all of OVS's features. We've only shown you a simple example here.

Now that you've seen one way of using network namespaces with OVS, let's look at a practical example: using containers with OVS.

Using containers with OVS

Because containers leverage network namespaces, a lot of what we discussed in the previous section applies here. The key differences are primarily in the container-specific workflow.

As of this writing, Docker containers did not have a built-in method for connecting containers to OVS for networking. Although Docker uses veth pairs and has the ability to use a Linux bridge, and although OVS has bridges that behave a lot like a Linux bridge, the glue to connect Docker containers to OVS did not materialize.

LXC, on the other hand, has built-in support for OVS. There are at least two ways to accomplish this:

- First, if you're using Libvirt with LXC, you can use a Libvirt virtual network to frontend an OVS bridge. We describe this process in the next section, Using VMs with OVS. The use of a Libvirt virtual network is identical, whether you're using containers or VMs.
- Alternatively, you can configure LXC to use a script to attach one of the veth interfaces to OVS.

Let's take a slightly closer look at that second option. (We're going to narrow our focus during this discussion to cover only LXC on Ubuntu.) We mentioned earlier that, by default, LXC stores container configuration information in */var/lib/lxc/<container-name>/config*, and it's in this file that you'll find the configuration options necessary to link LXC with OVS for networking. We covered a lot of these configuration options in Configuring LXC Networking, but there's one setting that is particularly applicable in this instance.

- The `lxc.network.script.up` option provides the name of a script that will be run when a container's network interface is set to up (enabled). Here is where you can provide a script that will take the veth pair (whose name is known, since it's controlled by the `lxc.network.veth.pair` directive) and attach it to an OVS bridge. A (simple) sample script might look something like this:

  ```
  #!/bin/bash

  BRIDGE="br0"
  ovs-vsctl --may-exist add-br $BRIDGE
  ovs-vsctl --if-exists del-port $BRIDGE $5
  ovs-vsctl --may-exist add-port $BRIDGE $5
  ```

The `$5` refers to the fifth parameter supplied to the script, which—in this specific case--is the name of the veth interface specified in the `lxc.network.veth.pair` configuration option. We haven't really discussed the --may-exist or --if-exists options to `ovs-vsctl`, but their behavior is just as you might expect. The --may-exist option prevents an error if the bridge or port already exists, while the --if -exists option takes an action only if the specified object exists.

Using this sort of configuration, LXC will create the veth pair (naming the interfaces according to the `lxc.network.veth.pair` configuration directive) and then run this script. The script will take the veth interface and attach it to the specified OVS bridge, and the container now has connectivity to OVS and whatever network topologies OVS is configured to support (bridged or overlay connectivity, for example).

What about using OVS with VMs? In the next section, you'll see that using VMs with OVS is generally also pretty straightforward.

Using VMs with OVS

To keep our discussion manageable, we'll focus (as we have in previous sections) on the KVM hypervisor with Libvirt. This is by no means a limit on OVS's part; it's simply a way for us to keep the amount of material manageable.

In Networking Virtual Machines, we introduced you to the concept of a Libvirt virtual network, which is an abstraction Libvirt uses to refer to lower-level constructs. For the last few years, Libvirt has offered built-in support for OVS, so that Libvirt virtual networks can leverage OVS directly.

The following bit of XML would define an OVS-backed Libvirt virtual network:

```
<network>
  <name>ovs-net</name>
  <forward mode="bridge"/>
  <bridge name="br0"/>
  <virtualport type="openvswitch"/>
</network>
```

You'd then reference this Libvirt virtual network by name in the KVM guest domain's configuration, like the following example (which shows only the networking-relevant portion of the guest domain's configuration):

```
<interface type="network">
  <source network="ovs-net"/>
</interface>
```

When using this sort of configuration, after the KVM guest domain is started you'll see a new interface attached to OVS when you run `ovs-vsctl show`:

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ovs-vsctl show</strong>
fe63a9ea-f72f-4aa2-b390-42ecbed6deef
    Bridge "br0"
        Port "ens3"
            Interface "ens3"
        Port "br0"
            Interface "br0"
                type: internal
        Port "vnet0"
            Interface "vnet0"
    ovs_version: "2.0.2"
ubuntu@ubuntu2004:~$
</pre>
```

This is a TAP interface, which you can verify with `ip -d link list vnet0` (note the "tun" in the output, which indicates it is a TUN/TAP device):

```
<pre data-type="programlisting">
```

```
ubuntu@ubuntu2004:~$ <strong>ip -d link list vnet0</strong>
7: vnet0: &lt;BROADCAST,MULTICAST,UP,LOWER_UP&gt; mtu 1500 qdisc pfifo_fast master
    ovs-system state UNKNOWN mode DEFAULT group default qlen 500
    link/ether fe:54:00:19:bc:6f brd ff:ff:ff:ff:ff:ff promiscuity 1
    tun
ubuntu@ubuntu2004:~$
</pre>
```

This VM is now bridged onto the physical network attached to ens3, but as with network namespaces you could leverage any of OVS's advanced features with this connection.

So far we've shown you connecting network namespaces, containers, and VMs to OVS. What if we want traffic from the host OVS system itself to flow through OVS? For that, you can use an OVS internal port.

Using OVS internal ports

OVS internal ports allow you to present a logical network interface to the host's TCP/IP stack. In that respect, you can compare OVS internal ports to VLAN interfaces, macvlan interfaces, or veth interfaces—all of these are logical network interfaces. The key difference here is that OVS internal ports *only* exist within the context of a particular OVS configuration.

Let's consider an example. We've shown you how to use an OVS bridge named br0 in the previous two sections. Every OVS bridge comes with a corresponding OVS internal port. You've seen this already, but you may not have noticed it. Consider this output of `ovs-vsctl show`:

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ovs-vsctl show</strong>
fe63a9ea-f72f-4aa2-b390-42ecbed6deef
    Bridge "br0"
        Port "eth1"
            Interface "eth1"
        Port "br0"
            Interface "br0"
                type: internal
    ovs_version: "2.0.2"
ubuntu@ubuntu2004:~$
</pre>
```

Note that br0 exists as a port, and as an interface with type internal. This is an OVS internal port, and the fact that `ip link list` shows the interface proves that the host's networking stack recognizes this as a logical network interface.

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ip link list br0</strong>
6: br0: &lt;BROADCAST,UP,LOWER_UP&gt; mtu 1500 qdisc noqueue state UNKNOWN mode DEFAULT
    group default
    link/ether 00:0c:29:7d:38:9d brd ff:ff:ff:ff:ff:ff
ubuntu@ubuntu2004:~$
</pre>
```

If you now delete the OVS bridge with `ovs-vsctl del-br br0`, what happens when we try to use `ip link list` to view the interface?

```
<pre data-type="programlisting">
ubuntu@ubuntu2004:~$ <strong>ip link list br0</strong>
Device "br0" does not exist.
ubuntu@ubuntu2004:~$
</pre>
```

This is what we mean when we say that an OVS internal port exists only within the context of an OVS configuration. It's not part of the host's network stack configuration; rather, it's part of the OVS configuration. Remove it from OVS, and it is removed from the host's network configuration.

You can use this to influence how the host's networking stack directs traffic. Let's say that you wanted to create a logical network interface that would serve as a tunnel endpoint (TEP) for VXLAN overlay traffic managed by OVS. Here are the commands you'd use to create an OVS internal port (we'll break this down after the example):

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ovs-vsctl add-port br0 tep0 -- set interface tep0
type=internal</strong>
[almalinux@alma9 ~]$ <strong>ovs-vsctl show</strong>
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    Bridge "br0"
        Port "br0"
            Interface "br0"
                type: internal
        Port "ens33"
            Interface "ens33"
        Port "tep0"
            Interface "tep0"
                type: internal
    ovs_version: "2.4.0"
</pre>
```

The unusual command syntax is needed because OVS expects interfaces to already exist when they are added to OVS. Naturally, tep0 doesn't exist, because we're creating it. So, we use the double-hyphen to tell OVS to link the commands together—thus creating the tep0 port and setting its type to internal at the same time.

Note that you *can* split the commands, if you don't mind OVS reporting an error first:

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ovs-vsctl add-port br0 tep0</strong>
ovs-vsctl: Error detected while setting up 'tep0'. See ovs-vswitchd log for details.
[almalinux@alma9 ~]$ <strong>ovs-vsctl set interface tep0 type=internal</strong>
[almalinux@alma9 ~]$ <strong>ovs-vsctl show</strong>
e1b45dda-69fa-4cb1-ad37-23eea2e63052
    Bridge "br0"
        Port "br0"
            Interface "br0"
```

```
                type: internal
        Port "ens33"
            Interface "ens33"
        Port "tep0"
            Interface "tep0"
                type: internal
    ovs_version: "2.4.0"
[almalinux@alma9 ~]$
</pre>
```

Now that the tep0 interface exists, you can configure it like you would any other logical interface. Here, we'll assign an IP address to the tep0 interface and set the interface to up (enabled):

```
<pre data-type="programlisting">
[almalinux@alma9 ~]$ <strong>ip link list tep0</strong>
10: tep0: &lt;BROADCAST,MULTICAST&gt; mtu 1500 qdisc noop state DOWN mode DEFAULT
    link/ether 9e:da:79:89:c3:6a brd ff:ff:ff:ff:ff:ff
[almalinux@alma9 ~]$ <strong>ip addr add 10.1.1.100/24 dev tep0</strong>
[almalinux@alma9 ~]$ <strong>ip link set tep0 up</strong>
[almalinux@alma9 ~]$ <strong>ip route list</strong>
default via 192.168.70.2 dev ens32  proto static  metric 100
10.1.1.0/24 dev tep0  proto kernel  scope link  src 10.1.1.100
192.168.70.0/24 dev ens32  proto kernel  scope link  src 192.168.70.244
192.168.70.0/24 dev ens32  proto kernel  scope link  src 192.168.70.244  metric
100
[almalinux@alma9 ~]$
</pre>
```

Based on the output of the `ip route list` command, you can see that the host's network configuration has been influenced by the configuration of the OVS internal port—this AlmaLinux system now has a new route associated with the IP address assigned to the tep0 interface.

Now let's see if you *really* understand how this configuration works: how does the traffic from tep0 get onto the network? If you said via the ens33 interface, you're exactly right! The OVS internal interface is a logical interface that is bridged onto the physical network via the br0 bridge, which contains the ens33 physical interface. Likewise, inbound traffic bound for 10.1.1.100/24 will enter the system via the ens33 interface.

This just barely scratches the surface of what is possible with OVS, but it should at least give you an idea of the basic concepts that are involved. As we mentioned earlier, OVS is a key part of a number of influential open source projects, so time spent working with OVS will pay off in a number of different areas.