

Chapter 4: Libvirt Networking

Understanding how virtual networking works is really essential for virtualization. It would be very hard to justify the costs associated with a scenario in which we didn't have virtual networking. Just imagine having multiple virtual machines on a virtualization host and buying network cards so that every single one of those virtual machines can have their own dedicated, physical network port. By implementing virtual networking, we're also consolidating networking in a much more manageable way, both from an administration and cost perspective.

This chapter provides you with an insight into the overall concept of virtualized networking and Linux-based networking concepts. We will also discuss physical and virtual networking concepts, try to compare them, and find similarities and differences between them. Also covered in this chapter is the concept of virtual switching for a per-host concept and spanned-across-hosts concept, as well as some more advanced topics. These topics include single-root input/output virtualization, which allows for a much more direct approach to hardware for certain scenarios. We will come back to some of the networking concepts later in this book as we start discussing

cloud overlay networks. This is because the basic networking concepts aren't scalable enough for large cloud environments.

In this chapter, we will cover the following topics:

- Understanding physical and virtual networking
- Using TAP/TUN
- Implementing Linux bridging
- Configuring Open vSwitch
- Understanding and configuring SR-IOV
- Understanding macvtap
- Let's get started!

Understanding physical and virtual networking

Let's think about networking for a second. This is a subject that most system administrators nowadays understand pretty well. This might not up to the level many of us think we do, but still – if we were to try to find an area of system administration where we'd find the biggest common level of knowledge, it would be networking.

So, what's the problem with that?

Actually, nothing much. If we really understand physical networking, virtual networking is going to be a piece of cake for us. Spoiler alert: *it's the same thing*. If we don't, it's going to be exposed

rather quickly, because there's no way around it. And the problems are going to get bigger and bigger as time goes by because environments evolve and – usually – grow. The bigger they are, the more problems they're going to create, and the more time you're going to spend in debugging mode.

That being said, if you have a firm grasp of VMware or Microsoft-based virtual networking purely at a technological level, you're in the clear here as all of these concepts are very similar.

With that out of the way, what's the whole hoopla about virtual networking? It's actually about understanding where things happen, how, and why. This is because, physically speaking, virtual networking is literally the same as physical networking. Logically speaking, there are some differences that relate more to the *topology* of things than to the principle or engineering side of things. And that's what usually throws people off a little bit – the fact that there are some weird, software-based objects that do the same job as the physical objects that most of us have grown used to managing via our favorite CLI-based or GUI-based utilities.

First, let's introduce the basic building block of virtualized networking – a virtual switch. A virtual switch is basically a software-based Layer 2 switch that you use to do two things:

- Hook up your virtual machines to it.

- Use its uplinks to connect them to physical server cards so that you can hook these physical network cards to a physical switch.

So, let's deal with why we need these virtual switches from the virtual machine perspective. As we mentioned earlier, we use a virtual switch to connect virtual machines to it. Why? Well, if we didn't have some kind of software object that sits in-between our physical network card and our virtual machine, we'd have a big problem – we could only connect virtual machines for which we have physical network ports to our physical network, and that would be intolerable. First, it goes against some of the basic principles of virtualization, such as efficiency and consolidation, and secondly, it would cost a lot. Imagine having 20 virtual machines on your server. This means that, without a virtual switch, you'd have to have at least 20 physical network ports to connect to the physical network. On top of that, you'd actually use 20 physical ports on your physical switch as well, which would be a disaster.

So, by introducing a virtual switch between a virtual machine and a physical network port, we're solving two problems at the same time – we're reducing the number of physical network adapters that we need per server, and we're reducing the number of physical switch ports that we need to use to connect our virtual machines to the network. We can actually argue that we're

solving a third problem as well – efficiency – as there are many scenarios where one physical network card can handle being an uplink for 20 virtual machines connected to a virtual switch. Specifically, there are large parts of our environments that don't consume a lot of network traffic and for those scenarios, virtual networking is just amazingly efficient.

Virtual networking

Now, in order for that virtual switch to be able to connect to something on a virtual machine, we have to have an object to connect to – and that object is called a virtual network interface card, often referred to as a vNIC. Every time you configure a virtual machine with a virtual network card, you're giving it the ability to connect to a virtual switch that uses a physical network card as an uplink to a physical switch.

Of course, there are some potential drawbacks to this approach. For example, if you have 50 virtual machines connected to the same virtual switch that uses the same physical network card as an uplink and that uplink fails (due to a network card issue, cable issue, switch port issue, or switch issue), your 50 virtual machines won't have access to the physical network. How do we get around this problem? By implementing a better design and following the basic design principles that we'd use on a physical network as well.

Specifically, we'd use more than one physical up-link to the same virtual switch.

Linux has *a lot* of different types of networking interfaces, something like 20 different types, some of which are as follows:

- **Bridge:** Layer 2 interface for (virtual machine) networking.
- **Bond:** For combining network interfaces to a single interface (for balancing and failover reasons) into one logical interface.
- **Team:** Different to bonding, teaming doesn't create one logical interface, but can still do balancing and failover.
- **MACVLAN:** Creates multiple MAC addresses on a single physical interface (creates subinterfaces) on Layer 2.
- **IPVLAN:** Unlike MACVLAN, IPVLAN uses the same MAC address and multiplexes on Layer 3.
- **MACVTAP/IPVTAP:** Newer drivers that should simplify virtual networking by combining TUN, TAP, and bridge as a single module.
- **VXLAN:** A commonly used cloud overlay network concept that we will describe in detail in [***Chapter 12***](#), *Scaling Out KVM with OpenStack*.
- **VETH:** A virtual Ethernet interface that can be used in a variety of ways for local tunneling.
- **IPOIB:** IP over Infiniband. As Infiniband gains traction in HPC/low latency networks, this type of networking is also supported by the Linux kernel.

There are a whole host of others. Then, on top of these network interface types, there are some 10 types of tunneling interfaces, some of which are as follows:

- **GRETAP, GRE:** Generic Routing Encapsulation protocols for encapsulating Layer 2 and Layer 3 protocols, respectively.
- **GENEVE:** A convergence protocol for cloud overlay networking that's meant to fuse VXLAN, GRE, and others into one. This is why it's supported in Open vSwitch, VMware NSX, and other products.
- **IPIP:** IP over IP tunnel for connecting internal IPv4 subnets via a public network.
- **SIT:** Simple Internet Translation for interconnecting isolated IPv6 networks over IPv4.
- **ip6tnl:** IPv4/6 tunnel over IPv6 tunnel interface.
- **IP6GRE, IP6GRETAP,** and others.

Getting your head around all of them is quite a complex and tedious process, so, in this book, we're only going to focus on the types of interfaces that are really important to us for virtualization and (later in this book) the cloud. This is why we will discuss VXLAN and GENEVE overlay networks in ***Chapter 12***, *Scaling Out KVM with OpenStack*, as we need to have a firm grip on **Software-Defined Networking (SDN)** as well.

So, specifically, as part of this chapter, we're going to cover TAP/TUN, bridging, Open vSwitch,

and macvtap interfaces as these are fundamentally the most important networking concepts for KVM virtualization.

But before we dig deep into that, let's explain a couple of basic virtual network concepts that apply to KVM/libvirt networking and other virtualization products (for example, VMware's hosted virtualization products such as Workstation or Player use the same concept). When you start configuring libvirt networking, you can choose between three basic types: NAT, routed, and isolated. Let's discuss what these networking modes do.

Libvirt NAT network

In a NAT libvirt network (and just to make sure that we mention this, the *default* network is configured like this), our virtual machine is behind a libvirt switch in NAT mode. Think of your *I have an internet connection @home scenario* – that's exactly what most of us have: our own *private* network behind a public IP address. This means that our device for accessing the internet (for example, DSL modem) connects to the public network (internet) and gets a public IP address as a part of that process. On our side of the network, we have our own subnet (for example, **192.168.0.0/24** or something like that) for all the devices that we want to connect to the internet.

Now, let's convert that into a virtualized network example. In our virtual machine scenario, this means that our virtual machine can communicate with anything that's connected to the physical network via host's IP address, but not the other way around. For something to communicate to our virtual machine behind a NAT'd switch, our virtual machine has to initiate that communication (or we have to set up some kind of port forwarding, but that's beside the point).

The following diagram might explain what we're talking about a bit better:

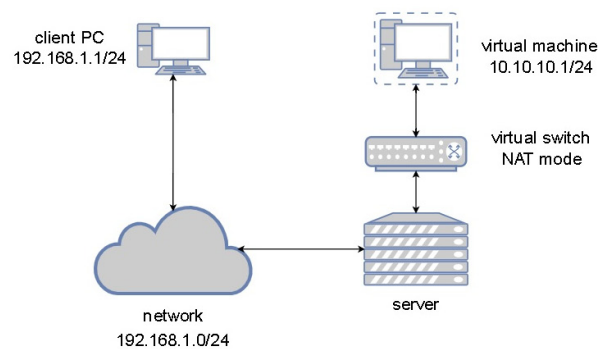


Figure 4.1 – libvirt networking in NAT mode

From the virtual machine perspective, it's happily sitting in a completely separate network segment (hence the **192.168.122.210** and **220** IP addresses) and using a virtual network switch as its gateway to access external networks. It doesn't have to be concerned with any kind of additional routing as that's one of the reasons why we use NAT – to simplify endpoint routing.

Libvirt routed network

The second network type is a routed network, which basically means that our virtual machine is directly connected to the physical network via a virtual switch. This means that our virtual machine is in the same Layer 2/3 network as the physical host. This type of network connection is used very often as, oftentimes, there is no need to have a separate NAT network to access your virtual machines in your environments. In a way, it just makes everything more complicated, especially because you have to configure routing to be aware of the NAT network that you're using for your virtual machines. When using routed mode, the virtual machine sits *in the same* network segment as the next physical device. The following diagram tells a thousand words about routed networks:

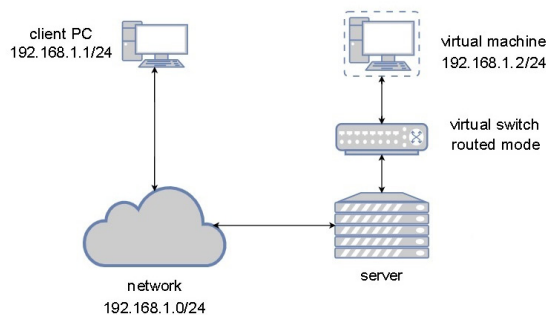


Figure 4.2 – libvirt networking in routed mode

Now that we've covered the two most commonly used types of virtual machine networking scenarios, it's time for the third one, which will seem a bit obscure. If we configure a virtual switch without any *uplinks* (which means it has no physical network cards attached to it), then that virtual switch can't send traffic to the physi-

cal network at all. All that's left is communication within the limits of that switch itself, hence the name *isolated*. Let's create that elusive isolated network now.

Libvirt isolated network

In this scenario, virtual machines attached to the same isolated switch can communicate with each other, but they cannot communicate with anything outside the host that they're running on. We used the word *obscure* to describe this scenario earlier, but it really isn't – in some ways, it's actually an ideal way of *isolating* specific types of traffic so that it doesn't even get to the physical network.

Think of it this way – let's say that you have a virtual machine that hosts a web server, for example, running WordPress. You create two virtual switches: one running routed network (direct connection to the physical network) and another that's isolated. Then, you can configure your WordPress virtual machine with two virtual network cards, with the first one connected to the routed virtual switch and the second one connected to the isolated virtual switch. WordPress needs a database, so you create another virtual machine and configure it to use an internal virtual switch only. Then, you use that isolated virtual switch to *isolate* traffic between the web server and the database server so that WordPress connects to the database server via

that switch. What did you get by configuring your virtual machine infrastructure like this? You have a two-tier application, and the most important part of that web application (database) is inaccessible from the outside world. Doesn't seem like that bad of an idea, right?

Isolated virtual networks are used in many other security-related scenarios, but this is just an example scenario that we can easily identify with.

Let's describe our isolated network with a diagram:

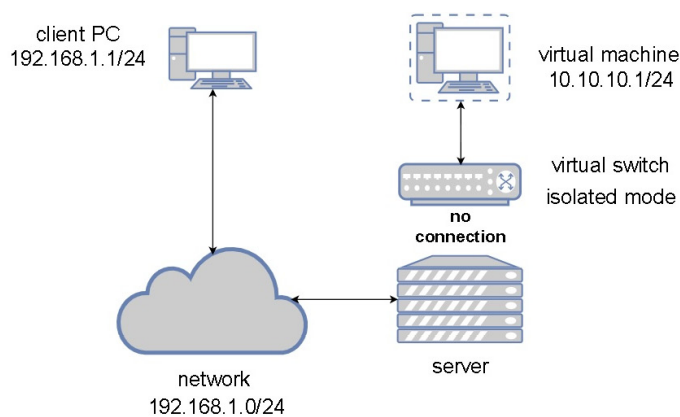


Figure 4.3 – libvirt networking in isolated mode

The previous chapter ([Chapter 3, Installing KVM Hypervisor, libvirt, and ovirt](#)) of this book mentioned the *default* network, and we said that we're going to talk about that a bit later. This seems like an opportune moment to do so because now, we have more than enough information to describe what the default network configuration is.

When we install all the necessary KVM libraries and utilities like we did in [Chapter 3, Installing](#)

KVM Hypervisor, libvirt, and oVirt, a default virtual switch gets configured out of the box. The reason for this is simple – it's more user-friendly to pre-configure something so that users can just start creating virtual machines and connecting them to the default network than expect users to configure that as well. VMware's vSphere hypervisor does the same thing (the default switch is called vSwitch0), and Hyper-V asks us during the deployment process to configure the first virtual switch (which we can actually skip and configure later). So, this is just a well-known, standardized, established scenario that enables us to start creating our virtual machines faster.

The default virtual switch works in NAT mode with the DHCP server active, and again, there's a simple reason for that – guest operating systems are, by default pre-configured with DHCP networking configuration, which means that the virtual machine that we just created is going to poll the network for necessary IP configuration. This way, the VM gets all the necessary network configuration and we can start using it right away.

The following diagram shows what the default KVM network does:

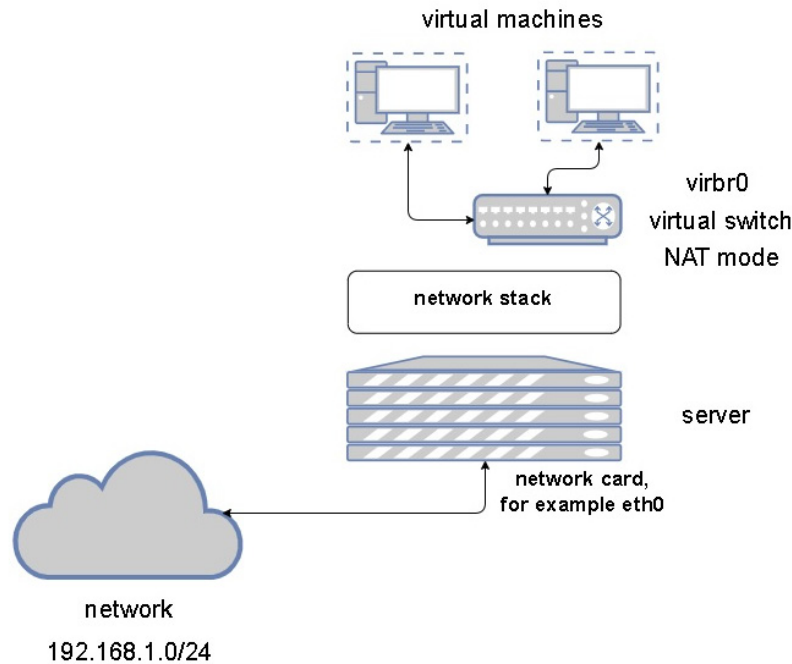


Figure 4.4 – libvirt default network in NAT mode

Now, let's learn how to configure these types of virtual networking concepts from the shell and from the GUI. We will treat this procedure as a procedure that needs to be done sequentially:

1. Let's start by exporting the default network configuration to XML so that we can use it as a template to create a new network:

```
[root@packtVM01 ~]# virsh net-dumpxml default > default.xml
[root@packtVM01 ~]# cat default.xml
<network>
  <name>default</name>
  <uuid>bb3fed90-ced1-45ce-ba3e-13c9ec64ff42</uuid>
  <forward mode='nat'>
    <nat>
      <port start='1024' end='65535' />
    </nat>
  </forward>
  <bridge name='virbr0' stp='on' delay='0' />
  <mac address='52:54:00:35:55:2d' />
  <ip address='192.168.122.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.122.2' end='192.168.122.254' />
    </dhcp>
  </ip>
</network>
```

Figure 4.5 – Exporting the default virtual network configuration

2. Now, let's copy that file to a new file called **packtnat.xml**, edit it, and then use it to create a new NAT virtual network. Before we do that, however, we need to generate two things – a

new object UUID (for our new network) and a unique MAC address. A new UUID can be generated from the shell by using the `uuidgen` command, but generating a MAC address is a bit trickier. So, we can use the standard Red Hat-proposed method available on the Red Hat website:

[https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/virtualization_administration/virtualization-tips_and_tricks-](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/virtualization_administration/virtualization-tips_and_tricks-generating_a_new_unique_mac_address)

[generating a new unique mac address](https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/virtualization_administration/virtualization-tips_and_tricks-generating_a_new_unique_mac_address). By using the first snippet of code available at that URL, create a new MAC address (for example, `00:16:3e:27:21:c1`).

By using `yum` command, install `python2`:

```
yum -y install python2
```

Make sure that you change the XML file so that it reflects the fact that we are configuring a new bridge (**virbr1**). Now, we can complete the configuration of our new virtual machine network XML file:

```
<network>
  <name>packtnat</name>
  <uuid>1f9e3fa3-859b-4bab-9598-d01d32336156</uuid>
  <forward mode='nat'>
    <nat>
      <port start='1024' end='65535' />
    </nat>
  </forward>
  <bridge name='virbr1' stp='on' delay='0' />
  <mac address='00:16:3e:27:21:c1' />
  <ip address='192.168.123.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.123.2' end='192.168.123.254' />
    </dhcp>
  </ip>
</network>
```

Figure 4.6 – New NAT network configuration

The next step is importing this configuration.

3. We can now use the `virsh` command to import that configuration and create our new virtual network, start that network and make it avail-

able permanently, and check if everything loaded correctly:

```
virsh net-define packtnat.xml
virsh net-start packtnat
virsh net-autostart packtnat
virsh net-list
```

Given that we didn't delete our default virtual network, the last command should give us the following output:

```
[root@packtVM01 ~]# virsh net-list
Name                State    Autostart    Persistent
-----
default             active   yes          yes
packtnat            active   yes          yes
```

Figure 4.7 – Using virsh net-list to check which virtual networks we have on the KVM host

Now, let's create two more virtual networks – a bridged network and an isolated network. Again, let's use files as templates to create both of these networks. Keep in mind that, in order to be able to create a bridged network, we are going to need a physical network adapter, so we need to have an available physical adapter in the server for that purpose. On our server, that interface is called **ens224**, while the interface called **ens192** is being used by the default libvirt network. So, let's create two configuration files called **packtro.xml** (for our routed network) and **packtiso.xml** (for our isolated network):


```

<network>
  <name>packtro</name>
  <uuid>3cac2e7a-a3fd-4b25-8717-450e665b7103</uuid>
  <forward dev='ens224' mode='route'>
    <interface dev='ens224' />
  </forward>
  <bridge name='virbr2' stp='on' delay='0' />
  <mac address='52:54:00:fe:9f:ec' />
  <domain name='packt' />
  <ip address='192.168.2.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.2.100' end='192.168.2.105' />
    </dhcp>
  </ip>
</network>

```

Figure 4.8 – libvirt routed network definition

In this specific configuration, we're using **ens224** as an uplink to the routed virtual network, which would use the same subnet (**192.168.2.0/24**) as the physical network that **ens224** is connected to:

```

<network>
  <name>packtiso</name>
  <uuid>2b92b03d-acb4-4a23-b205-2095c6a27bd4</uuid>
  <bridge name='virbr3' stp='on' delay='0' />
  <mac address='00:16:3e:0b:5d:85' />
</bridge>
  <domain name='packtiso' />
  <ip address='192.168.3.1' netmask='255.255.255.0'>
    <dhcp>
      <range start='192.168.3.128' end='192.168.3.254' />
    </dhcp>
  </ip>
</network>

```

Figure 4.9 – libvirt isolated network definition

Just to cover our bases, we could have easily configured all of this by using the Virtual Machine Manager GUI, as that application has a wizard for creating virtual networks as well. But when we're talking about larger environments, importing XML is a much simpler process, even when we forget about the fact that a lot of KVM virtualization hosts don't have a GUI installed at all.

So far, we've discussed virtual networking from an overall host-level. However, there's also a different approach to the subject – using a virtual machine as an object to which we can add a virtual network card and connect it to a virtual network. We can use **virsh** for that purpose. So, just

as an example, we can connect our virtual machine called **MasteringKVM01** to an isolated virtual network:

```
virsh attach-interface --domain  
MasteringKVM01 --source isolated --  
type network --model virtio --config  
--live
```

There are other concepts that allow virtual machine connectivity to a physical network, and some of them we will discuss later in this chapter (such as SR-IOV). However, now that we've covered the basic approaches to connecting virtual machines to a physical network by using a virtual switch/bridge, we need to get a bit more technical. The thing is, there are more concepts involved in connecting a virtual machine to a virtual switch, such as TAP and TUN, which we will be covering in the following section.

Using userspace networking with TAP and TUN devices

In ***Chapter 1, Understanding Linux***

Virtualization, we used the **virt-host-validate** command to do some pre-flight checks in terms of the host's preparedness for KVM virtualization. As a part of that process, some of the checks include checking if the following devices exist:

- **/dev/kvm**: The KVM drivers create a **/dev/kvm** character device on the host to facilitate direct hardware access for virtual machines. Not having this device means that the VMs won't be able to access physical hardware, although it's enabled in the BIOS and this will reduce the VM's performance significantly.
- **/dev/vhost-net**: The **/dev/vhost-net** character device will be created on the host. This device serves as the interface for configuring the **vhost-net** instance. Not having this device significantly reduces the virtual machine's network performance.
- **/dev/net/tun**: This is another character special device used for creating TUN/TAP devices to facilitate network connectivity for a virtual machine. The TUN/TAP device will be explained in detail in future chapters. For now, just understand that having a character device is important for KVM virtualization to work properly.

Let's focus on the last device, the TUN device, which is usually accompanied by a TAP device.

So far, all the concepts that we've covered include some kind of connectivity to a physical network card, with isolated virtual networks being an exception. But even an isolated virtual network is just a virtual network for our virtual machines. What happens when we have a situation where we need our communication to happen in the user space, such as between applications

running on a server? It would be useless to patch them through some kind of virtual switch concept, or a regular bridge, as that would just bring additional overhead. This is where TUN/TAP devices come in, providing packet flow for user space programs. Easily enough, an application can open `/dev/net/tun` and use an `ioctl()` function to register a network device in the kernel, which, in turn, presents itself as a `tunXX` or `tapXX` device. When the application closes the file, the network devices and routes created by it disappear (as described in the kernel `tuntap.txt` documentation). So, it's just a type of virtual network interface for the Linux operating system supported by the Linux kernel – you can add an IP address and routes to it so that traffic from your application can route through it, and not via a regular network device.

TUN emulates an L3 device by creating a communication tunnel, something like a point-to-point tunnel. It gets activated when the `tuntap` driver gets configured in `tun` mode. When you activate it, any data that you receive from a descriptor (the application that configured it) will be data in the form of regular IP packages (as the most commonly used case). Also, when you send data, it gets written to the TUN device as regular IP packages. This type of interface is sometimes used in testing, development, and debugging for simulation purposes.

The TAP interface basically emulates an L2 Ethernet device. It gets activated when the tun-tap driver gets configured in tap mode. When you activate it, unlike what happens with the TUN interface (Layer 3), you get Layer 2 raw Ethernet packages, including ARP/RARP packages and everything else. Basically, we're talking about a virtualized Layer 2 Ethernet connection.

These concepts (especially TAP) are usable on libvirt/QEMU as well because by using these types of configurations, we can create connections from the host to a virtual machine – without the libvirt bridge/switch, just as an example. We can actually configure all of the necessary details for the TUN/TAP interface and then start deploying virtual machines that are hooked up directly to those interfaces by using `kvm-qemu` options. So, it's a rather interesting concept that has its place in the virtualization world as well. This is especially interesting when we start creating Linux bridges.

Implementing Linux bridging

Let's create a bridge and then add a TAP device to it. Before we do that, we must make sure the bridge module is loaded into the kernel. Let's get started:

1. If it is not loaded, use **modprobe bridge** to load the module:

```
# lsmod | grep bridge
```

Run the following command to create a bridge called **tester**:

```
# brctl addbr tester
```

Let's see if the bridge has been created:

```
# brctl show  
bridge name bridge id STP enabled  
interfaces  
tester 8000.460a80dd627d no
```

The **# brctl show** command will list all the available bridges on the server, along with some basic information, such as the ID of the bridge, **Spanning Tree Protocol (STP)** status, and the interfaces attached to it. Here, the tester bridge does not have any interfaces attached to its virtual ports.

2. A Linux bridge will also be shown as a network device. To see the network details of the bridge tester, use the **ip** command:

```
# ip link show tester  
6: tester: <BROADCAST,MULTICAST>mtu  
1500 qdiscnoop state DOWN mode  
DEFAULT group default link/ether  
26:84:f2:f8:09:e0  
brdff:ff:ff:ff:ff:ff
```

You can also use **ifconfig** to check and configure the network settings for a Linux bridge; **ifconfig** is relatively easy to read and understand but not as feature-rich as the **ip** command:

```
# ifconfig tester
tester:
flags=4098<BROADCAST,MULTICAST>mtu
1500
ether26:84:f2:f8:09:e0txqueuelen
1000 (Ethernet)
RX packets 0 bytes 0 (0.0 B)
RX errors 0 dropped 0 overruns 0
frame 0
TX packets 0 bytes 0 (0.0 B)
TX errors 0 dropped 0 overruns 0
carrier 0 collisions 0
```

The Linux bridge tester is now ready. Let's create and add a TAP device to it.

3. First, check if the TUN/TAP device module is loaded into the kernel. If not, you already know the drill:

```
# lsmod | greptun
tun 28672 1
```

Run the following command to create a tap device named **vm-vnic**:

```
# ip tuntap add dev vm-vnic mode
tap
# ip link show vm-vnic
7: vm-vnic:
<BROADCAST,MULTICAST>mtu 1500
qdiscnoop state DOWN
mode DEFAULT group default qlen 500
link/ether 46:0a:80:dd:62:7d
brdff:ff:ff:ff:ff:ff
```

We now have a bridge named **tester** and a tap device named **vm-vnic**. Let's add **vm-vnic** to

tester:

```
# brctl addif tester vm-vnic
# brctl show
bridge name bridge id STP enabled
interfaces
tester 8000.460a80dd627d no vm-vnic
```

Here, you can see that **vm-vnic** is an interface that was added to the **tester** bridge. Now, **vm-vnic** can act as the interface between your virtual machine and the **tester** bridge, which, in turn, enables the virtual machine to communicate with other virtual machines that are added to this bridge:

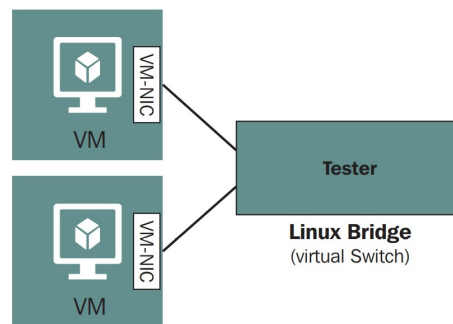


Figure 4.10 – Virtual machines connected to a virtual switch (bridge)

You might also need to remove all the objects and configurations that were created in the previous procedure. Let's do this step by step via the command line:

1. First, we need to remove the **vm-vnic** tap device from the **tester** bridge:

```
# brctl delif tester vm-vnic
# brctl show tester
```



```
bridge name bridge id STP enabled  
interfaces  
tester 8000.460a80dd627d no
```

Once the **vm-vnic** has been removed from the bridge, remove the tap device using the **ip** command:

```
# ip tuntap del dev vm-vnic mode  
tap
```

2. Then, remove the tester bridge:

```
# brctl delbr tester
```

These are the same steps that libvirt carried out in the backend while enabling or disabling networking for a virtual machine. We want you to understand this procedure thoroughly before moving ahead. Now that we've covered Linux bridging, it's time to move on to a more advanced concept called Open vSwitch.

Configuring Open vSwitch

Imagine for a second that you're working for a small company that has three to four KVM hosts, a couple of network-attached storage devices to host their 15 virtual machines, and that you've been employed by the company from the very start. So, you've seen it all – the company buying some servers, network switches, cables, and storage devices, and you were a part of a small team of people that built that environment. After 2 years of that process, you're aware of the fact

that everything works, it's simple to maintain, and doesn't give you an awful lot of grief.

Now, imagine the life of a friend of yours working for a bigger enterprise company that has 400 KVM hosts and close to 2,000 virtual machines to manage, doing the same job as you're doing in a comfy chair of your office in your small company.

Do you think that your friend can manage his or her environment by using the very same tools that you're using? XML files for network switch configuration, deploying servers from a bootable USB drive, manually configuring everything, and having the time to do so? Does that seem like a possibility to you?

There are two basic problems in this second situation:

- The scale of the environment: This one is more obvious. Because of the environment size, you need some kind of concept that's going to be managed centrally, instead of on a host-per-host level, such as the virtual switches we've discussed so far.
- Company policies: These usually dictate some kind of compliance that comes from configuration standardization as much as possible. Now, we can agree that we could script some configuration updates via Ansible, Puppet, or something like that, but what's the use? We're going to have to create new config files, new proce-

dures, and new workbooks every single time we need to introduce a change to KVM networking. And big companies frown upon that.

So, what we need is a centralized networking object that can span across multiple hosts and offer configuration consistency. In this context, configuration consistency offers us a huge advantage – every change that we introduce in this type of object will be replicated to all the hosts that are members of this centralized networking object. In other words, what we need is **Open vSwitch (OVS)**. For those who are more versed in VMware-based networking, we can use an approximate metaphor – Open vSwitch is for KVM-based environments similar to what vSphere Distributed Switch is for VMware-based environments.

In terms of technology, OVS supports the following:

- VLAN isolation (IEEE 802.1Q)
- Traffic filtering
- NIC bonding with or without LACP
- Various overlay networks – VXLAN, GENEVE, GRE, STT, and so on
- 802.1ag support
- Netflow, sFlow, and so on
- (R)SPAN
- OpenFlow
- OVSDB
- Traffic queuing and shaping

- Linux, FreeBSD, NetBSD, Windows, and Citrix support (and a host of others)

Now that we've listed some of the supported technologies, let's discuss the way in which Open vSwitch works.

First, let's talk about the Open vSwitch architecture. The implementation of Open vSwitch is broken down into two parts: the Open vSwitch kernel module (the data plane) and the user space tools (the control pane). Since the incoming data packets must be processed as fast as possible, the data plane of Open vSwitch was pushed to the kernel space:

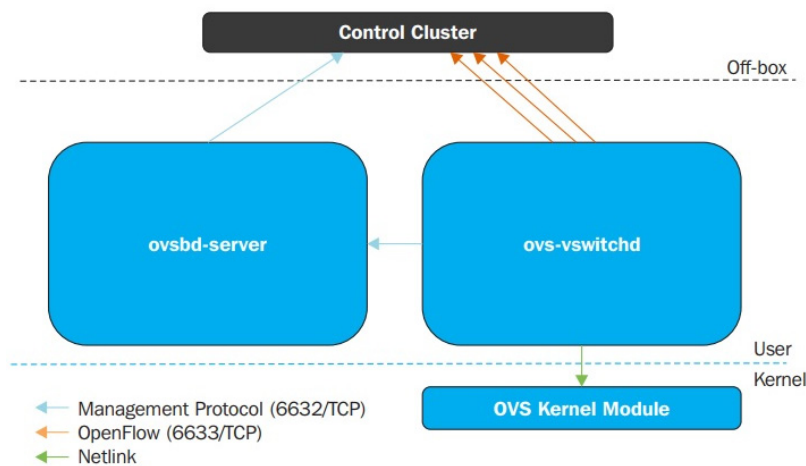


Figure 4.11 – Open vSwitch architecture

The data path (OVS kernel module) uses the netlink socket to interact with the vswitchd daemon, which implements and manages any number of OVS switches on the local system.

Open vSwitch doesn't have a specific SDN controller that it uses for management purposes, in a similar fashion to VMware's vSphere distrib-

uted switch and NSX, which have vCenter and various NSX components to manage their capabilities. In OVS, the point is to use someone else's SDN controller, which then interacts with `ovs-vswitchd` using the OpenFlow protocol. The `ovsdb-server` maintains the switch table database and external clients can talk to the `ovsdb-server` using JSON-RPC; JSON is the data format. The `ovsdb` database currently contains around 13 tables and this database is persistent across restarts.

Open vSwitch works in two modes: normal and flow mode. This chapter will primarily concentrate on how to bring up a KVM VM connected to Open vSwitch's bridge in standalone/normal mode and will give a brief introduction to flow mode using the OpenDaylight controller:

- **Normal Mode:** Switching and forwarding are handled by OVS bridge. In this mode OVS acts as an L2 learning switch. This mode is specifically useful when configuring several overlay networks for your target rather than manipulating the switch's flow.
- **Flow Mode:** In flow mode, the Open vSwitch bridge flow table is used to decide on which port the receiving packets should be forwarded to. All the flows are managed by an external SDN controller. Adding or removing the control flow requires using an SDN controller that's managing the bridge or using the `ctl` command. This mode allows a greater level of

abstraction and automation; the SDN controller exposes the REST API. Our applications can make use of this API to directly manipulate the bridge's flows to meet network needs.

Let's move on to the practical aspect and learn how to install Open vSwitch on CentOS 8:

1. The first thing that we must do is tell our system to use the appropriate repositories. In this case, we need to enable the repositories called **epel** and **centos-release-openstack-train**. We can do that by using a couple of **yum** commands:

```
yum -y install epel-release  
yum -y install centos-release-  
openstack-train
```

2. The next step will be installing **openvswitch** from Red Hat's repository:

```
dnf install openvswitch -y
```

3. After the installation process, we need to check if everything is working by starting and enabling the Open vSwitch service and running the **ovs-vsctl -V** command:

```
systemctl start openvswitch  
systemctl enable openvswitch  
ovs-vsctl -V
```

The last command should throw you some output specifying the version of Open vSwitch and its DB schema. In our case, it's Open vSwitch **2.11.0** and DB schema **7.16.1**.

4. Now that we've successfully installed and started Open vSwitch, it's time to configure it.

Let's choose a deployment scenario in which we're going to use Open vSwitch as a new virtual switch for our virtual machines. In our server, we have another physical interface called **ens256**, which we're going to use as an uplink for our Open vSwitch virtual switch. We're also going to clear ens256 configuration, configure an IP address for our OVS, and start the OVS by using the following commands:

```
ovs-vsctl add-br ovs-br0
ip addr flush dev ens256
ip addr add 10.10.10.1/24 dev ovs-br0
ovs-vsctl add-port ovs-br0 ens256
ip link set dev ovs-br0 up
```

5. Now that everything has been configured but not persistently, we need to make the configuration persistent. This means configuring some network interface configuration files. So, go to **/etc/sysconfig/network-scripts** and create two files. Call one of them **ifcfg-ens256** (for our uplink interface):

```
DEVICE=ens256
TYPE=OVSPort
DEVICETYPE=ovs
OVS_BRIDGE=ovs-br0
ONBOOT=yes
```

Call the other file **ifcfg-ovs-br0** (for our OVS):

```
DEVICE=ovs-br0
DEVICETYPE=ovs
TYPE=OVSBridge
BOOTPROTO=static
```

```
IPADDR=10.10.10.1
NETMASK=255.255.255.0
GATEWAY=10.10.10.254
ONBOOT=yes
```

6. We didn't configure all of this just for show, so we need to make sure that our KVM virtual machines are also able to use it. This means – again – that we need to create a KVM virtual network that's going to use OVS. Luckily, we've dealt with KVM virtual network XML files before (check the *Libvirt isolated network* section), so this one isn't going to be a problem. Let's call our network **packtovs** and its corresponding XML file **packtovs.xml**. It should contain the following content:

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

So, now, we can perform our usual operations when we have a virtual network definition in an XML file, which is to define, start, and autostart the network:

```
virsh net-define packtovs.xml
virsh net-start packtovs
virsh net-autostart packtovs
```

If we left everything as it was when we created our virtual networks, the output from **virsh net-list** should look something like this:

Name	State	Autostart	Persistent
default	active	yes	yes
packtiso	active	yes	yes
packtnat	active	yes	yes
packtovs	active	yes	yes
packtro	active	yes	yes

Figure 4.12 – Successful OVS configuration, and OVS+KVM configuration

So, all that's left now is to hook up a VM to our newly defined OVS-based network called **packtovs** and we're home free. Alternatively, we could just create a new one and pre-connect it to that specific interface using the knowledge we gained in [Chapter 3, Installing KVM Hypervisor, libvirt, and oVirt](#). So, let's issue the following command, which has just two changed parameters (**--name** and **--network**):

```
virt-install --virt-type=kvm --name
MasteringKVM03 --vcpus 2 --ram 4096 -
-os-variant=rhel8.0 --
cdrom=/var/lib/libvirt/images/CentOS-
8-x86_64-1905-dvd1.iso --network
network:packtovs --graphics vnc --
disk size=16
```

After the virtual machine installation completes, we're connected to the OVS-based **packtovs** virtual network, and our virtual machine can use it. Let's say that additional configuration is needed and that we got a request to tag traffic coming from this virtual machine with **VLAN ID 5**. Start your virtual machine and use the following set of commands:

```
ovs-vsctl list-ports ovs-br0
ens256
```

```
vnet0
```

This command tells us that we're using the **ens256** port as an uplink and that our virtual machine, **MasteringKVM03**, is using the virtual **vnet0** network port. We can apply VLAN tagging to that port by using the following command:

```
ovs-vsctl set port vnet0 tag=5
```

We need to take note of some additional commands related to OVS administration and management since this is done via the CLI. So, here are some commonly used OVS CLI administration commands:

- **#ovs-vsctl show**: A very handy and frequently used command. It tells us what the current running configuration of the switch is.
- **#ovs-vsctl list-br**: Lists bridges that were configured on Open vSwitch.
- **#ovs-vsctl list-ports <bridge>**: Shows the names of all the ports on **BRIDGE**.
- **#ovs-vsctl list interface <bridge>**: Shows the names of all the interfaces on **BRIDGE**.
- **#ovs-vsctl add-br <bridge>**: Creates a bridge in the switch database.
- **#ovs-vsctl add-port <bridge> : <interface>**: Binds an interface (physical or virtual) to the Open vSwitch bridge.
- **#ovs-ofctl** and **ovs-dpctl**: These two commands are used for administering and monitoring flow entries. You learned that OVS manages two kinds of flows: OpenFlows and Datapath. The first is managed in the control

plane, while the second one is a kernel-based flow.

- **#ovs-ofctl**: This speaks to the OpenFlow module, whereas **ovs-dpctl** speaks to the Kernel module.

The following examples are the most used options for each of these commands:

- **#ovs-ofctl show <BRIDGE>**: Shows brief information about the switch, including the port number to port name mapping.
- **#ovs-ofctl dump-flows <Bridge>**: Examines OpenFlow tables.
- **#ovs-dpctl show**: Prints basic information about all the logical datapaths, referred to as *bridges*, present on the switch.
- **#ovs-dpctl dump-flows**: It shows the flow cached in datapath.
- **ovs-appctl**: This command offers a way to send commands to a running Open vSwitch and gathers information that is not directly exposed to the **ovs-ofctl** command. This is the Swiss Army knife of OpenFlow troubleshooting.
- **#ovs-appctl bridge/dumpflows
**: Examines flow tables and offers direct connectivity for VMs on the same hosts.
- **#ovs-appctl fdb/show
**: Lists MAC/VLAN pairs learned.

Also, you can always use the **ovs-vsctl show** command to get information about the configu-

ration of your OVS switch:

```
[root@packtVM01 ~]# ovs-vsctl show
c64c1381-af64-4b51-8db7-f62e37319a06
  Bridge "ovs-br0"
    Port "ovs-br0"
      Interface "ovs-br0"
        type: internal
    Port "vnet0"
      tag: 0
      Interface "vnet0"
    Port "ens256"
      Interface "ens256"
  ovs version: "2.11.0"
```

Figure 4.13 – ovs-vsctl show output

We are going to come back to the subject of Open vSwitch in ***Chapter 12, Scaling Out KVM with OpenStack***, as we go deeper into our discussion about spanning Open vSwitch across multiple hosts, especially while keeping in mind the fact that we want to be able to span our cloud overlay networks (based on GENEVE, VXLAN, GRE, or similar protocols) across multiple hosts and sites.

Other Open vSwitch use cases

As you might imagine, Open vSwitch isn't just a handy concept for libvirt or OpenStack – it can be used for a variety of other scenarios as well. Let's describe one of them as it might be important for people looking into VMware NSX or NSX-T integration.

Let's just describe a few basic terms and relationships here. VMware's NSX is an SDN-based technology that can be used for a variety of use cases:

- Connecting data centers and extending cloud overlay networks across data center boundaries.

- A variety of disaster recover scenarios. NSX can be a big help for disaster recover, for multi-site environments, and for integration with a variety of external services and devices that can be a part of the scenario (Palo Alto PANs).
- Consistent micro-segmentation, across sites, done *the right way* on the virtual machine network card level.
- For security purposes, varying from different types of supported VPN technologies to connect sites and end users, to distributed firewalls, guest introspection options (antivirus and anti-malware), network introspection options (IDS/IPS), and more.
- For load balancing, up to Layer 7, with SSL off-load, session persistence, high availability, application rules, and more.

Yes, VMware's take on SDN (NSX) and Open vSwitch seem like *competing technologies* on the market, but realistically, there are loads of clients who want to use both. This is where VMware's integration with OpenStack and NSX's integration with Linux-based KVM hosts (by using Open vSwitch and additional agents) comes in really handy. Just to further explain these points – there are things that NSX does that take *extensive* usage of Open vSwitch-based technologies – hardware VTEP integration via Open vSwitch Database, extending GENEVE networks

to KVM hosts by using Open vSwitch/NSX integration, and much more.

Imagine that you're working for a service provider – a cloud service provider, an ISP; basically, any type of company that has large networks with a lot of network segmentation. There are loads of service providers using VMware's vCloud Director to provide cloud services to end users and companies. However, because of market needs, these environments often need to be extended to include AWS (for additional infrastructure growth scenarios via the public cloud) or OpenStack (to create hybrid cloud scenarios). If we didn't have a possibility to have interoperability between these solutions, there would be no way to use both of these offerings at the same time. But from a networking perspective, the network background for that is NSX or NSX-T (which actually *uses* Open vSwitch).

It's been clear for years that the future is all about multi-cloud environments, and these types of integrations will bring in more customers; they will want to take advantage of these options in their cloud service design. Future developments will also most probably include (and already partially include) integration with Docker, Kubernetes, and/or OpenShift to be able to manage containers in the same environment.

There are also some more extreme examples of using hardware – in our example, we are talking

about network cards on a PCI Express bus – in a *partitioned* way. For the time being, our explanation of this concept, called SR-IOV, is going to be limited to network cards, but we will expand on the same concept in [Chapter 6](#), *Virtual Display Devices and Protocols*, when we start talking about partitioning GPUs for use in virtual machines. So, let's discuss a practical example of using SR-IOV on an Intel network card that supports it.

Understanding and using SR-IOV

The SR-IOV concept is something that we already mentioned in [Chapter 2](#), *KVM as a Virtualization Solution*. By utilizing SR-IOV, we can *partition* PCI resources (for example, network cards) into virtual PCI functions and inject them into a virtual machine. If we're using this concept for network cards, we're usually doing this with a single purpose – so that we can avoid using the operating system kernel and network stack while accessing a network interface card from our virtual machine. In order for us to be able to do this, we need to have hardware support, so we need to check if our network card actually supports it. On a physical server, we could use the **lspci** command to extract attribute information about our PCI devices and then **grep** out *Single Root I/O Virtualization* as a string to try to see if we have a

device that's compatible. Here's an example from our server:

```
root@storage3:~# lspci -s 24:00 -vvv | grep -i Single
Capabilities: [160 v1] Single Root I/O Virtualization (SR-IOV)
Capabilities: [160 v1] Single Root I/O Virtualization (SR-IOV)
```

Figure 4.14 – Checking if our system is SR-IOV compatible

Important Note

Be careful when configuring SR-IOV. You need to have a server that supports it, a device that supports it, and you must make sure that you turn on SR-IOV functionality in BIOS. Then, you need to keep in mind that there are servers that only have specific slots assigned for SR-IOV. The server that we used (HP Proliant DL380p G8) has three PCI-Express slots assigned to CPU1, but SR-IOV worked only in slot #1. When we connected our card to slot #2 or #3, we got a BIOS message that SR-IOV will not work in that slot and that we should move our card to a slot that supports SR-IOV. So, please, make sure that you read the documentation of your server thoroughly and connect a SR-IOV compatible device to a correct PCI-Express slot.

In this specific case, it's an Intel 10 Gigabit network adapter with two ports, which we could use to do SR-IOV. The procedure isn't all that difficult, and it requires us to complete the following steps:

1. Unbind from the previous module.

2. Register it to the `vfio-pci` module, which is available in the Linux kernel stack.
3. Configure a guest that's going to use it.

So, what you would do is unload the module that the network card is currently using by using `modprobe -r`. Then, you would load it again, but by assigning an additional parameter. On our specific server, the Intel dual-port adapter that we're using (X540-AT2) was assigned to the `ens1f0` and `ens1f1` network devices. So, let's use `ens1f0` as an example for SR-IOV configuration at boot time:

1. The first thing that we need to do (as a general concept) is find out which kernel module our network card is using. To do that, we need to issue the following command:

```
ethtool -i ens1f0 | grep ^driver
```

In our case, this is the output that we got:

```
driver: ixgbe
```

We need to find additional available options for that module. To do that, we can use the `modinfo` command (we're only interested in the `parm` part of the output):

```
modinfo ixgbe  
  
...  
  
Parm: max_vfs (Maximum number of  
virtual functions to allocate per  
physical function - default is zero  
and maximum value is 63.
```

For example, we're using the `ixgbe` module here, and we can do the following:

```
modprobe -r ixgbe  
modprobe ixgbe max_vfs=4
```

2. Then, we can use the **modprobe** system to make these changes permanent across reboots by creating a file in **/etc/modprobe.d** called (for example) **ixgbe.conf** and adding the following line to it:

```
options ixgbe max_vfs=4
```

This would give us up to four virtual functions that we can use inside our virtual machines. Now, the next issue that we need to solve is how to boot our server with SR-IOV active at boot time. There are quite a few steps involved here, so, let's get started:

1. We need to add the **iommu** and **vfs** parameters to the default kernel boot line and the default kernel configuration. So, first, open **/etc/default/grub** and edit the **GRUB_CMDLINE_LINUX** line and add **intel_iommu=on** (or **amd_iommu=on** if you're using an AMD system) and **ixgbe.max_vfs=4** to it.
2. We need to reconfigure **grub** to use this change, so we need to use the following command:

```
grub2-mkconfig -o  
/boot/grub2/grub.cfg
```

3. Sometimes, even that isn't enough, so we need to configure the necessary kernel parameters, such as the maximum number of virtual functions and the **iommu** parameter to be used on

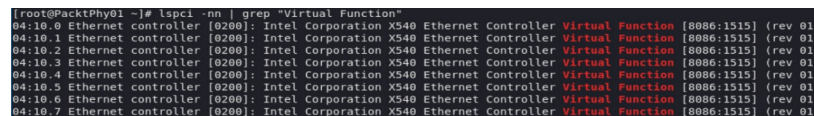
the server. That leads us to the following command:

```
grubby --update-kernel=ALL --
args="intel_iommu=on
ixgbe.max_vfs=4"
```

After reboot, we should be able to see our virtual functions. Type in the following command:

```
lspci -nn | grep "Virtual Function"
```

We should get an output that looks like this:



```
[root@PacktPhy01 ~]# lspci -nn | grep "Virtual Function"
04:10.0 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.1 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.2 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.3 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.4 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.5 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.6 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
04:10.7 Ethernet controller [0200]: Intel Corporation X540 Ethernet Controller Virtual Function [8086:1515] (rev 01)
```

Figure 4.15 – Checking for virtual function visibility

We should be able to see these virtual functions from libvirt, and we can check that via the **virsh** command. Let's try this (we're using **grep 04** because our device IDs start with 04, which is visible from the preceding image; we'll shrink the output to important entries only):

```
virsh nodedev-list | grep 04
.....
pci_0000_04_00_0
pci_0000_04_00_1
pci_0000_04_10_0
pci_0000_04_10_1
pci_0000_04_10_2
pci_0000_04_10_3
pci_0000_04_10_4
pci_0000_04_10_5
```

```
pci_0000_04_10_6
```

```
pci_0000_04_10_7
```

The first two devices are our physical functions. The remaining eight devices (two ports times four functions) are our virtual devices (from `pci_0000_04_10_0` to `pci_0000_04_10_7`). Now, let's dump that device's information by using the `virsh nodedev-dumpxml pci_0000_04_10_0` command:

```
[root@PacktPhy01 ~]# virsh nodedev-dumpxml pci_0000_04_10_0
<device>
  <name>pci_0000_04_10_0</name>
  <path>/sys/devices/pci0000:00/0000:00:03.0/0000:04:10.0</path>
  <parent>pci_0000_00_03_0</parent>
  <driver>
    <name>ixgbevf</name>
  </driver>
  <capability type='pci'>
    <domain>0</domain>
    <bus>4</bus>
    <slot>16</slot>
    <function>0</function>
    <product id='0x1515'>X540 Ethernet Controller Virtual Function</product>
    <vendor id='0x8086'>Intel Corporation</vendor>
    <capability type='phys_function'>
      <address domain='0x0000' bus='0x04' slot='0x00' function='0x0'>
    </capability>
    <iommuGroup number='60'>
      <address domain='0x0000' bus='0x04' slot='0x10' function='0x0'>
    </iommuGroup>
    <numa node='0'>
    </numa>
    <pci-express>
      <link validity='cap' port='0' speed='5' width='8'>
    </link>
      <link validity='sta' width='0'>
    </link>
    </pci-express>
  </capability>
</device>
```

Figure 4.16 – Virtual function information from the perspective of virsh

So, if we have a running virtual machine that we'd like to reconfigure to use this, we'd have to create an XML file with definition that looks something like this (let's call it `packtsriov.xml`):

```
<interface type='hostdev'
  managed='yes' >
  <source>
    <address type='pci'
      domain='0x0000' bus='0x04'
      slot='0x10' function='0x0'>
    </address>
```

```
</source>  
</interface>
```

Of course, the domain, bus, slot, and function need to point exactly to our VF. Then, we can use the **virsh** command to attach that device to our virtual machine (for example, **MasteringKVM03**):

```
virsh attach-device MasteringKVM03  
packtsriov.xml --config
```

When we use **virsh dumpxml**, we should now see a part of the output that starts with **<driver name='vfio' />**, along with all the information that we configured in the previous step (address type, domain, bus, slot, function). Our virtual machine should have no problems using this virtual function as a network card.

Now, it's time to cover another concept that's very much useful in KVM networking: **macvtap**. It's a newer driver that should simplify our virtualized networking by completely removing tun/tap and bridge drivers with a single module.

Understanding macvtap

This module works like a combination of the tap and macvlan modules. We already explained what the tap module does. The macvlan module enables us to create virtual networks that are pinned to a physical network interface (usually, we call this interface a *lower* interface or device). Combining tap and macvlan enables us to choose between four different modes of operation,

called **Virtual Ethernet Port Aggregator (VEPA)**, bridge, private, and passthru.

If we're using the VEPA mode (default mode), the physical switch has to support VEPA by supporting **hairpin** mode (also called reflective relay). When a *lower* device receives data from a VEPA mode macvlan, this traffic is always sent out to the upstream device, which means that traffic is always going through an external switch. The advantage of this mode is the fact that network traffic between virtual machines becomes visible on the external network, which can be useful for a variety of reasons. You can check how network flow works in the following sequence of diagrams:

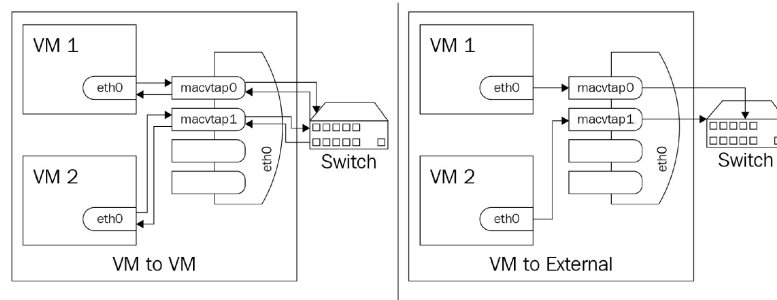


Figure 4.17 – macvtap VEPA mode, where traffic is forced to the external network

In private mode, it's similar to VEPA in that everything goes to an external switch, but unlike VEPA, traffic only gets delivered if it's sent via an external router or switch. You can use this mode if you want to isolate virtual machines connected to the endpoints from one another, but not from the external network. If this sounds very much

like a private VLAN scenario, you're completely correct:

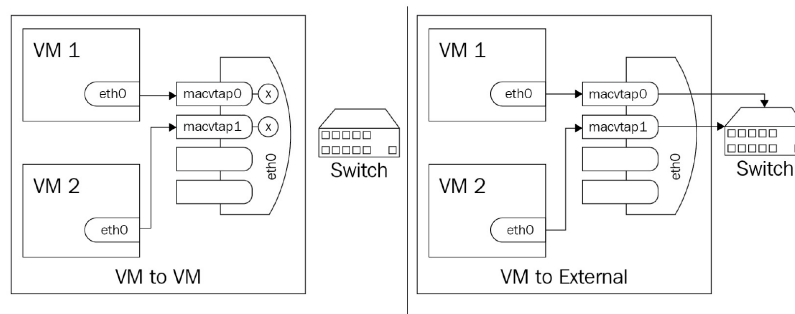


Figure 4.18 – macvtap in private mode, using it for internal network isolation

In bridge mode, data received on your macvlan that's supposed to go to another macvlan on the same lower device is sent directly to the target, not externally, and then routed back. This is very similar to what VMware NSX does when communication is supposed to happen between virtual machines on different VXLAN networks, but on the same host:

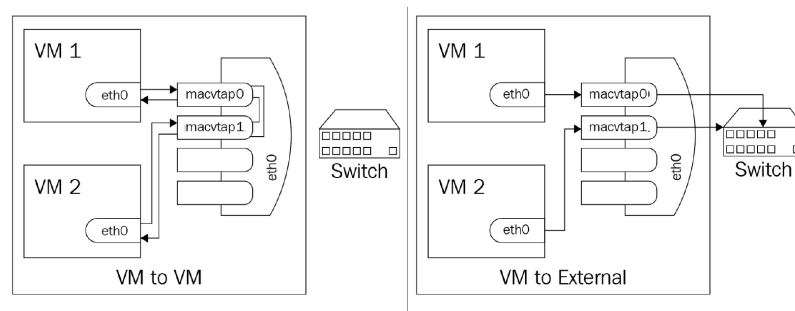


Figure 4.19 – macvtap in bridge mode, providing a kind of internal routing

In passthrough mode, we're basically talking about the SR-IOV scenario, where we're using a VF or a physical device directly to the macvtap interface. The key difference is that a single net-

work interface can only be passed to a single guest (1:1 relationship):

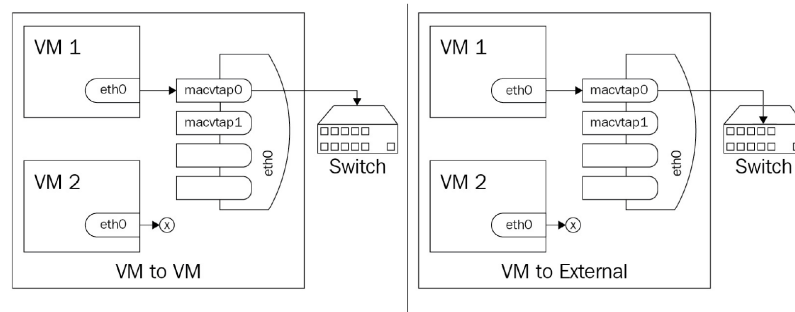


Figure 4.20 – macvtap in passthrough mode

In [Chapter 12](#), *Scaling Out KVM with OpenStack* and [Chapter 13](#), *Scaling Out KVM with AWS*, we'll describe why virtualized and *overlay* networking (VXLAN, GRE, GENEVE) is even more important for cloud networking as we extend our local KVM-based environment to the cloud either via OpenStack or AWS.

Summary

In this chapter, we covered the basics of virtualized networking in KVM and explained why virtualized networking is such a huge part of virtualization. We went knee-deep into configuration files and their options as this will be the preferred method for administration in larger environments, especially when talking about virtualized networks.

Pay close attention to all the configuration steps that we discussed through this chapter, especially the part related to using `virsh` commands to manipulate network configuration and to con-

figure Open vSwitch and SR-IOV. SR-IOV-based concepts are heavily used in latency-sensitive environments to provide networking services with the lowest possible overhead and latency, which is why this principle is very important for various enterprise environments related to the financial and banking sector.

Now that we've covered all the necessary networking scenarios (some of which will be revisited later in this book), it's time to start thinking about the next big topic of the virtualized world. We've already talked about CPU and memory, as well as networks, which means we're left with the fourth pillar of virtualization: storage. We will tackle that subject in the next chapter.

Questions

1. Why is it important that virtual switches accept connectivity from multiple virtual machines at the same time?
2. How does a virtual switch work in NAT mode?
3. How does a virtual switch work in routed mode?
4. What is Open vSwitch and for what purpose can we use it in virtualized and cloud environments?
5. Describe the differences between TAP and TUN interfaces.

Further reading

Please refer to the following links for more information regarding what was covered in this chapter:

- Libvirt networking:
<https://wiki.libvirt.org/page/VirtualNetworking>
- Network XML format:
<https://libvirt.org/formatnetwork.html>
- Open vSwitch: <https://www.openvswitch.org/>
- Open vSwitch and libvirt:
<http://docs.openvswitch.org/en/latest/howto/libvirt/>
- Open vSwitch Cheat Sheet:
<https://adhioutlined.github.io/virtual/Openvswitch-Cheat-Sheet/>