

Chapter 5: Libvirt Storage

This chapter provides you with an insight into the way that KVM uses storage. Specifically, we will cover both storage that's internal to the host where we're running virtual machines and *shared storage*. Don't let the terminology confuse you here – in virtualization and cloud technologies, the term *shared storage* means storage space that multiple hypervisors can have access to. As we will explain a bit later, the three most common ways of achieving this are by using block-level, share-level, or object-level storage. We will use NFS as an example of share-level storage, and **Internet Small Computer System Interface (iSCSI)** and **Fiber Channel (FC)** as examples of block-level storage. In terms of object-based storage, we will use Ceph. GlusterFS is also commonly used nowadays, so we'll make sure that we cover that, too. To wrap everything up in an easy-to-use and easy-to-manage box, we will discuss some open source projects that might help you while practicing with and creating testing environments.

In this chapter, we will cover the following topics:

- Introduction to storage
- Storage pools
- NFS storage
- iSCSI and SAN storage
- Storage redundancy and multipathing

- Gluster and Ceph as a storage backend for KVM
- Virtual disk images and formats and basic KVM storage operations
- The latest developments in storage – NVMe and NVMeOF

Introduction to storage

Unlike networking, which is something that most IT people have at least a basic understanding of, storage tends to be quite different. In short, yes, it tends to be a bit more complex. There are loads of parameters involved, different technologies, and...let's be honest, loads of different types of configuration options and people enforcing them. And a *lot* of questions. Here are some of them:

- Should we configure one NFS share per storage device or two?
- Should we create one iSCSI target per storage device or two?
- Should we create one FC target or two?
- How many **Logical Unit Numbers (LUNs)** per target?
- What kind of cluster size should we use?
- How should we carry out multipathing?
- Should we use block-level or share-level storage?
- Should we use block-level or object-level storage?
- Which technology or solution should we choose?

- How should we configure caching?
- How should we configure zoning or masking?
- How many switches should we use?
- Should we use some kind of clustering technology on a storage level?

As you can see, the questions just keep piling up, and we've barely touched the surface, because there are also questions about which filesystem to use, which physical controller we will use to access storage, and what type of cabling—it just becomes a big mashup of variables that has many potential answers. What makes it worse is the fact that many of those answers can be correct—not just one of them.

Let's get the basic-level mathematics out of the way. In an enterprise-level environment, shared storage is usually *the most expensive* part of the environment and can also have *the most significant negative impact* on virtual machine performance, while at the same time being *the most oversubscribed resource* in that environment. Let's think about this for a second—every powered-on virtual machine is constantly going to hammer our storage device with I/O operations. If we have 500 virtual machines running on a single storage device, aren't we asking a bit too much from that storage device?

At the same time, some kind of shared storage concept is a key pillar of virtualized environments. The basic principle is very simple – there are loads of advanced functionalities that will work so much better with shared storage. Also,

many operations are much faster if shared storage is available. Even more so, there are so many simple options for high availability when we don't have our virtual machines stored in the same place where they are being executed.

As a bonus, we can easily avoid **Single Point Of Failure (SPOF)** scenarios if we design our shared storage environment correctly. In an enterprise-level environment, avoiding SPOF is one of the key design principles. But when we start adding switches and adapters and controllers to the *to buy* list, our managers' or clients' heads usually starts to hurt. We talk about performance and risk management, while they talk about price. We talk about the fact that their databases and applications need to be properly fed in terms of I/O and bandwidth, and they feel that you can produce that out of thin air. Just wave your magic wand and there we are: unlimited storage performance.

But the best, and our all-time favorite, apples-to-oranges comparison that your clients are surely going to try to enforce on you goes something like this...*"the shiny new 1 TB NVMe SSD in my laptop has more than 1,000 times more IOPS and more than 5 times more performance than your \$50,000 storage device, while costing 100 times less! You have no idea what you're doing!"*

If you've been there, we feel for you. Rarely will you see so many discussions and fights about a piece of hardware in a box. But it's such an essential piece of hardware in a box that it's a good

fight to have. So, let's explain some key concepts that libvirt uses in terms of storage access and how to work with it. Then, let's use our knowledge to extract as much performance as possible out of our storage system and libvirt using it.

In this chapter, we're basically going to cover almost all of these storage types via installation and configuration examples. Each and every one of these has its own use case, but generally, it's going to be up to you to choose what you're going to use.

So, let's start our journey through these supported protocols and learn how to configure them. After we cover storage pools, we are going to discuss NFS, a typical share-level protocol for virtual machine storage. Then, we're going to move to block-level protocols such as iSCSI and FC. Then, we will move to redundancy and multipathing to increase the availability and bandwidth of our storage devices. We're also going to cover various use cases for not-so-common filesystems (such as Ceph, Gluster, and GFS) for KVM virtualization. We're also going to discuss the new developments that are de facto trends right now.

Storage pools

When you first start using storage devices—even if they're cheaper boxes—you're faced with some choices. They will ask you to do a bit of configuration—select the RAID level, configure hot-spares, SSD caching...it's a process. The same

process applies to a situation in which you're building a data center from scratch or extending an existing one. You have to configure the storage to be able to use it.

Hypervisors are a bit *picky* when it comes to storage, as there are storage types that they support and storage types that they don't support. For example, Microsoft's Hyper-V supports SMB shares for virtual machine storage, but it doesn't really support NFS storage for virtual machine storage. VMware's vSphere Hypervisor supports NFS, but it doesn't support SMB. The reason is simple—a company developing a hypervisor chooses and qualifies technologies that its hypervisor is going to support. Then, it's up to various HBA/controller vendors (Intel, Mellanox, QLogic, and so on) to develop drivers for that hypervisor, and it's up to storage vendor to decide which types of storage protocols they're going to support on their storage device.

From a CentOS perspective, there are many different storage pool types that are supported. Here are some of them:

- **Logical Volume Manager (LVM)**-based storage pools
- Directory-based storage pools
- Partition-based storage pools
- GlusterFS-based storage pools
- iSCSI-based storage pools
- Disk-based storage pools
- HBA-based storage pools, which use SCSI devices

From the perspective of libvirt, a storage pool can be a directory, a storage device, or a file that libvirt manages. That leads us to 10+ different storage pool types, as you're going to see in the next section. From a virtual machine perspective, libvirt manages virtual machine storage, which virtual machines use so that they have the capacity to store data.

oVirt, on the other hand, sees things a bit differently, as it has its own service that works with libvirt to provide centralized storage management from a data center perspective. *Data center perspective* might seem like a term that's a bit odd. But think about it—a datacenter is some kind of *higher-level* object in which you can see all of your resources. A data center uses *storage* and *hypervisors* to provide us with all of the services that we need in virtualization—virtual machines, virtual networks, storage domains, and so on. Basically, from a data center perspective, you can see what's happening on all of your hosts that are members of that datacenter. However, from a host level, you can't see what's happening on another host. It's a hierarchy that's completely logical from both a management and a security perspective.

oVirt can centrally manage these different types of storage pools (and the list can get bigger or smaller as the years go by):

- **Network File System (NFS)**
- **Parallel NFS (pNFS)**
- **iSCSI**

- FC
- Local storage (attached directly to KVM hosts)
- GlusterFS exports
- POSIX-compliant file systems

Let's take care of some terminology first:

- **Brtfs** is a type of filesystem that supports snapshots, RAID and LVM-like functionality, compression, defragmentation, online resizing, and many other advanced features. It was deprecated after it was discovered that its RAID5/6 can easily lead to a loss of data.
- **ZFS** is a type of filesystem that supports everything that Brtfs does, plus read and write caching.

CentOS has a new way of dealing with storage pools. Although still in technology preview state, it's worth going through the complete configuration via this new tool, called **Stratis**. Basically, a couple of years ago, Red Hat finally deprecated the idea of pushing Brtfs for future releases and started working on Stratis. If you've ever used ZFS, that's where this is probably going—an easy-to-manage, ZFS-like, volume-managing set of utilities that Red Hat can stand behind in their future releases. Also, just like ZFS, a Stratis-based pool can use cache; so, if you have an SSD that you'd like to dedicate to pool cache, you can actually do that, as well. If you have been expecting Red Hat to support ZFS, there's a fundamental Red Hat policy that stands in the way. Specifically, ZFS is not a part of the Linux kernel, mostly because of licensing reasons. Red Hat has

a policy for these situations—if it's not a part of the kernel (upstream), then they don't provide nor support it. As it stands, that's not going to happen anytime soon. These policies are also reflected in CentOS.

Local storage pools

On the other hand, Stratis is available right now. We're going to use it to manage our local storage by creating storage pools. Creating a pool requires us to set up partitions or disks beforehand. After we create a pool, we can create a volume on top of it. We only have to be very careful about one thing—although Stratis can manage XFS filesystems, we shouldn't make changes to Stratis-managed XFS filesystems directly from the filesystem level. For example, do not reconfigure or reformat a Stratis-based XFS filesystem directly from XFS-based commands because you'll create havoc on your system.

Stratis supports various different types of block storage devices:

- Hard disks and SSDs
- iSCSI LUNs
- LVM
- LUKS
- MD RAID
- A device mapper multipath
- NVMe devices

Let's start from scratch and install Stratis so that we can use it. Let's use the following command:

```
yum -y install stratisd stratis-cli  
systemctl enable --now stratisd
```

The first command installs the Stratis service and the corresponding command-line utilities. The second one will start and enable the Stratis service.

Now, we are going to go through a complete example of how to use Stratis to configure your storage devices. We're going to cover an example of this layered approach. So, what we are going to do is as follows:

- Create a software RAID10 + spare by using MD RAID.
- Create a Stratis pool out of that MD RAID device.
- Add a cache device to the pool to use Stratis' cache capability.
- Create a Stratis filesystem and mount it on our local server.

The premise here is simple—the software RAID10+ spare via MD RAID is going to approximate the regular production approach, in which you'd have some kind of a hardware RAID controller presenting a single block device to the system. We're going to add a cache device to the pool to verify the caching functionality, as this is something that we would most probably do if we were using ZFS, as well. Then, we are going to create a filesystem on top of that pool and mount it to a local directory with the help of the following commands:

```
mdadm --create /dev/md0 --verbose --  
level=10 --raid-devices=4 /dev/sdb  
/dev/sdc /dev/sdd /dev/sde --spare-  
devices=1 /dev/sdf2  
stratis pool create  
PacktStratisPool01 /dev/md0  
stratis pool add-cache  
PacktStratisPool01 /dev/sdg  
stratis pool add-cache  
PacktStratisPool01 /dev/sdg  
stratis fs create PackStratisPool01  
PacktStratisXFS01  
mkdir /mnt/packtStratisXFS01  
mount  
/stratis/PacktStratisPool01/PacktStra  
tisXFS01 /mnt/packtStratisXFS01
```

This mounted filesystem is XFS-formatted. We could then easily use this filesystem via NFS export, which is exactly what we're going to do in the NFS storage lesson. But for now, this was just an example of how to create a pool by using Stratis.

We've covered some basics of local storage pools, which brings us closer to our next subject, which is how to use pools from a libvirt perspective. So, that will be our next topic.

Libvirt storage pools

Libvirt manages its own storage pools, which is done with one thing in mind—to provide different pools for virtual machine disks and related data. Keeping in mind that libvirt uses what the underlying operating system supports, it's no

wonder that it supports loads of different storage pool types. A picture is worth a thousand words, so here's a screenshot of creating a libvirt storage pool from virt-manager:

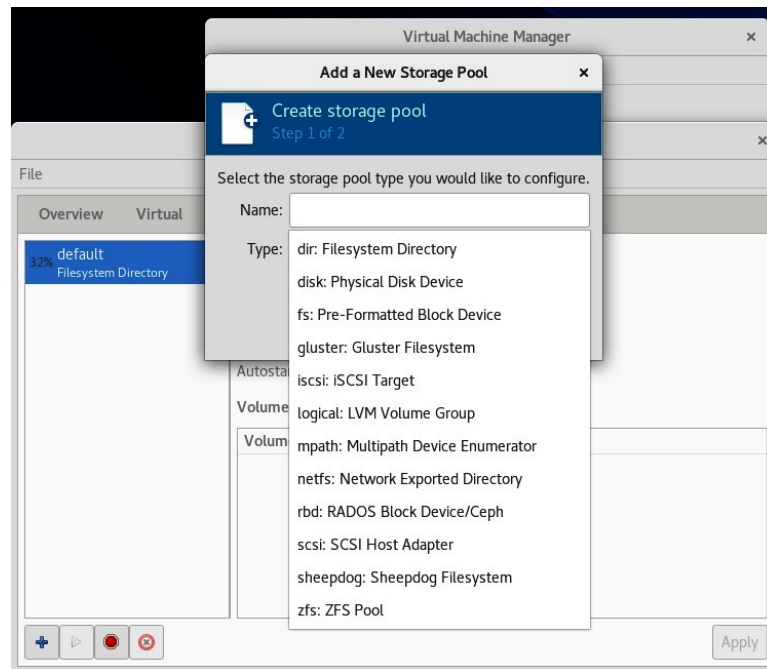


Figure 5.1 – Different storage pool types supported by libvirt

Out of the box, libvirt already has a predefined default storage pool, which is a directory storage pool on the local server. This default pool is located in the `/var/lib/libvirt/images` directory. This represents our default location where we'll save all the data from locally installed virtual machines.

We're going to create various different types of storage pools in the following sections—an NFS-based pool, an iSCSI and FC pool, and Gluster and Ceph pools: the whole nine yards. We're also going to explain when to use each and every one of them as there will be different usage models involved.

NFS storage pool

As a protocol, NFS has been around since the mid-80s. It was originally developed by Sun Microsystems as a protocol for sharing files, which is what it's been used for up to this day. Actually, it's still being developed, which is quite surprising for a technology that's so *old*. For example, NFS version 4.2 came out in 2016. In this version, NFS received a very big update, such as the following:

- **Server-side copy:** A feature that significantly enhances the speed of cloning operations between NFS servers by carrying out cloning directly between NFS servers
- **Sparse files and space reservation:** Features that enhance the way NFS works with files that have unallocated blocks, while keeping an eye on capacity so that we can guarantee space availability when we need to write data
- **Application data block support:** A feature that helps applications that work with files as block devices (disks)
- Better pNFS implementation

There are other bits and pieces that were enhanced in v4.2, but for now, this is more than enough. You can find even more information about this in IETF's RFC 7862 document (<https://tools.ietf.org/html/rfc7862>). We're going to focus our attention on the implementation of NFS v4.2 specifically, as it's the best that NFS

currently has to offer. It also happens to be the default NFS version that CentOS 8 supports.

The first thing that we have to do is install the necessary packages. We're going to achieve that by using the following commands:

```
yum -y install nfs-utils  
systemctl enable --now nfs-server
```

The first command installs the necessary utilities to run the NFS server. The second one is going to start it and permanently enable it so that the NFS service is available after reboot.

Our next task is to configure what we're going to share via the NFS server. For that, we need to *export* a directory and make it available to our clients over the network. NFS uses a configuration file, **/etc/exports**, for that purpose. Let's say that we want to create a directory called **/exports**, and then share it to our clients in the **192.168.159.0/255.255.255.0** network, and we want to allow them to write data on that share. Our **/etc/exports** file should look like this:

```
/mnt/packtStratisXFS01  
192.168.159.0/24(rw)  
exportfs -r
```

These configuration options tell our NFS server which directory to export (**/exports**), to which clients (**192.168.159.0/24**), and what options to use (**rw** means read-write).

Some other available options include the following:

- **ro**: Read-only mode.
- **sync**: Synchronous I/O operations.
- **root_squash**: All I/O operations from **UID 0** and **GID 0** are mapped to configurable anonymous UIDs and GIDs (the **anonuid** and **anongid** options).
- **all_squash**: All I/O operations from any UIDs and GIDs are mapped to anonymous UIDs and GIDs (**anonuid** and **anongid** options).
- **no_root_squash**: All I/O operations from **UID 0** and **GID 0** are mapped to **UID 0** and **GID 0**.

If you need to apply multiple options to the exported directory, you add them with a comma between them, as follows:

```
/mnt/packtStratisXFS01  
192.168.159.0/24(rw, sync, root_squash)
```

You can use fully qualified domain names or short hostnames (if they're resolvable by DNS or any other mechanism). Also, if you don't like using prefixes (**24**), you can use regular netmasks, as follows:

```
/mnt/packtStratisXFS01  
192.168.159.0/255.255.255.0(rw, root_squash)
```

Now that we have configured the NFS server, let's see how we're going to configure libvirt to use that server as a storage pool. As always, there are a couple of ways to do this. We could just create an XML file with the pool definition and import it to our KVM host by using the **virsh pool-define --file** command. Here's an example of that configuration file:

```

<pool type='netfs'>
  <name>NFSpool1</name>
  <source>
    <host name='192.168.159.144' />
    <dir path='/mnt/packtStratisXFS01' />
    <format type='auto' />
  </source>
  <target>
    <path>/var/lib/libvirt/images/NFSpool1</path>
    <permissions>
      <mode>0755</mode>
      <owner>0</owner>
      <group>0</group>
      <label>system_u:object_r:nfs_t:s0</label>
    </permissions>
  </target>
</pool>

```

Figure 5.2 – Example XML configuration file for NFS pool

Let's explain these configuration options:

- **pool type:** `netfs` means that we are going to use an NFS file share.
- **name:** The pool name, as libvirt uses pools as named objects, just like virtual networks.
- **host :** The address of the NFS server that we are connecting to.
- **dir path:** The NFS export path that we configured on the NFS server via `/etc/exports`.
- **path:** The local directory on our KVM host where that NFS share is going to be mounted to.
- **permissions:** The permissions used for mounting this filesystem.
- **owner and group:** The UID and GID used for mounting purposes (that's why we exported the folder earlier with the `no_root_squash` option).
- **label:** The SELinux label for this folder—we're going to discuss this in [*Chapter 16, Troubleshooting Guideline for the KVM Platform*](#).

If we wanted, we could've easily done the same thing via the Virtual Machine Manager GUI. First, we would have to select the correct type (the NFS pool) and give it a name:

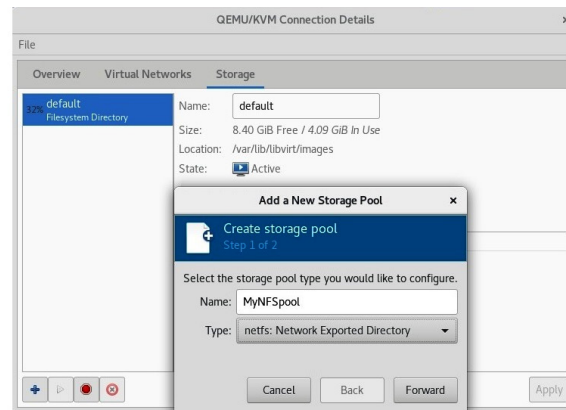


Figure 5.3 – Selecting the NFS pool type and giving it a name

After we click **Forward**, we can move to the final configuration step, where we need to tell the wizard which server we're mounting our NFS share from:

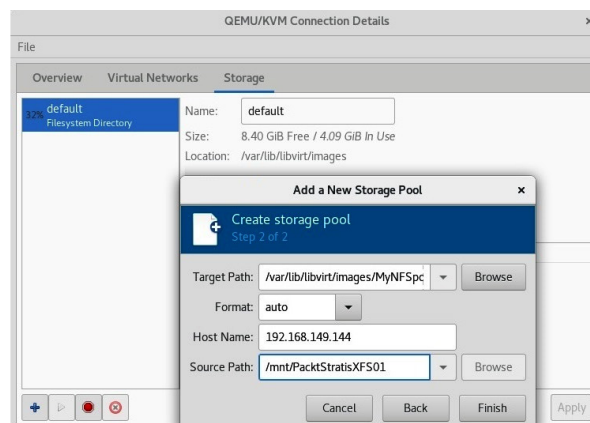


Figure 5.4 – Configuring NFS server options

When we finish typing in these configuration options (**Host Name** and **Source Path**), we can press **Finish**, which will mean exiting the wizard. Also, our previous configuration screen, which only contained the **default** storage pool,

now has our newly configured pool listed as well:

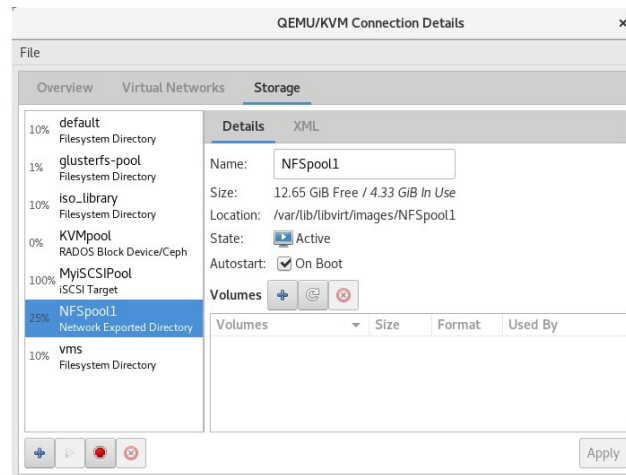


Figure 5.5 – Newly configured NFS pool visible on the list

When would we use NFS-based storage pools in libvirt, and for what? Basically, we can use them nicely for anything related to the storage of installation images—ISO files, virtual floppy disk files, virtual machine files, and so on.

Please remember that even though it seemed that NFS is almost gone from enterprise environments just a while ago, NFS is still around. Actually, with the introduction of NFS 4.1, 4.2, and pNFS, its future on the market actually looks even better than a couple of years ago. It's such a familiar protocol with a very long history, and it's still quite competitive in many scenarios. If you're familiar with VMware virtualization technology, VMware introduced a technology called Virtual Volumes in ESXi 6.0. This is an object-based storage technology that can use both block- and NFS-based protocols for its basis, which is a really compelling use case for some

scenarios. But for now, let's move on to block-level technologies, such as iSCSI and FC.

iSCSI and SAN storage

Using iSCSI for virtual machine storage has long been the regular thing to do. Even if you take into account the fact that iSCSI isn't the most efficient way to approach storage, it's still so widely accepted that you'll find it everywhere.

Efficiency is compromised for two reasons:

- iSCSI encapsulates SCSI commands into regular IP packages, which means segmentation and overhead as IP packages have a pretty large header, which means less efficiency.
- Even worse, it's TCP-based, which means that there are sequence numbers and retransmissions, which can lead to queueing and latency, and the bigger the environment is, the more you usually feel these effects affect your virtual machine performance.

That being said, the fact that it's based on an Ethernet stack makes it easier to deploy iSCSI-based solutions, while at the same time offering some unique challenges. For example, sometimes it's difficult to explain to a customer that using the same network switch(es) for virtual machine traffic and iSCSI traffic is not the best idea. What makes it even worse is the fact that clients are sometimes so blinded by their desire to save money that they don't understand that they're working against their own best interest. Especially when it comes to network bandwidth.

Most of us have been there, trying to work with clients' questions such as "*but we already have a Gigabit Ethernet switch, why would you need anything faster than that?*"

The fact of the matter is, with iSCSI's intricacies, more is just – more. The more speed you have on the disk/cache/controller side and the more bandwidth you have on the networking side, the more chance you have of creating a storage system that's faster. All of that can have a big impact on our virtual machine performance. As you'll see in the *Storage redundancy and multipathing* section, you can actually build a very good storage system yourself—both for iSCSI and FC. This might come in real handy when you try to create some kind of a testing lab/environment to play with as you develop your KVM virtualization skills. You can apply that knowledge to other virtualized environments, as well.

The iSCSI and FC architectures are very similar—they both need a target (an iSCSI target and an FC target) and an initiator (an iSCSI initiator and an FC initiator). In this terminology, the target is a *server* component, and the initiator is a *client* component. To put it simply, the initiator connects to a target to get access to block storage that's presented via that target. Then, we can use the initiator's identity to *limit* what the initiator is able to see on the target. This is where the terminology starts to get a bit different when comparing iSCSI and FC.

In iSCSI, the initiator's identity can be defined by four different properties. They are as follows:

- **iSCSI Qualified Name (IQN):** This is a unique name that all initiators and targets have in iSCSI communication. We can compare this to a MAC or IP address in regular Ethernet-based networks. You can think of it this way—an IQN is for iSCSI what a MAC or IP address is for Ethernet-based networks.
- **IP address:** Every initiator will have a different IP address that it uses to connect to the target.
- **MAC address:** Every initiator has a different MAC address on Layer 2.
- **Fully Qualified Domain Name (FQDN):** This represents the name of the server as it's resolved by a DNS service.

From the iSCSI target perspective—depending on its implementation—you can use any one of these properties to create a configuration that's going to tell the iSCSI target which IQNs, IP addresses, MAC addresses, or FQDNs can be used to connect to it. This is what's called *masking*, as we can *mask* what an initiator can *see* on the iSCSI target by using these identities and pairing them with LUNs. LUNs are just raw, block capacities that we export via an iSCSI target toward initiators. LUNs are *indexed*, or *numbered*, usually from 0 onward. Every LUN number represents a different storage capacity that an initiator can connect to.

For example, we can have an iSCSI target with three different LUNs—**LUN0** with 20 GB, **LUN1** with 40 GB, and **LUN2** with 60 GB. These will all be hosted on the same storage system's iSCSI target. We can then configure the iSCSI target to accept an IQN to see all the LUNs, another IQN to only see **LUN1**, and another IQN to only see **LUN1** and **LUN2**. This is actually what we are going to configure right now.

Let's start by configuring the iSCSI target service. For that, we need to install the **targetcli** package, and configure the service (called **target**) to run:

```
yum -y install targetcli  
systemctl enable --now target
```

Be careful about the firewall configuration; you might need to configure it to allow connectivity on port **3260/tcp**, which is the port that the iSCSI target portal uses. So, if your firewall has started, type in the following command:

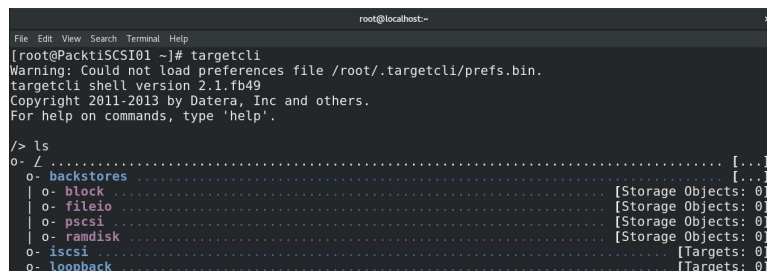
```
firewall-cmd --permanent --add-  
port=3260/tcp ; firewall-cmd --reload
```

There are three possibilities for iSCSI on Linux in terms of what storage backend to use. We could use a regular filesystem (such as XFS), a block device (a hard drive), or LVM. So, that's exactly what we're going to do. Our scenario is going to be as follows:

- **LUN0** (20 GB): XFS-based filesystem, on the **/dev/sdb** device

- **LUN1 (40 GB):** Hard drive, on the **/dev/sdc** device
- **LUN2 (60 GB):** LVM, on the **/dev/sdd** device

So, after we install the necessary packages and configure the target service and firewall, we should start with configuring our iSCSI target. We'll just start the **targetcli** command and check the state, which should be a blank slate as we're just beginning the process:



```

root@localhost~
[root@PacktiSCSI01 ~]# targetcli
Warning: Could not load preferences file /root/.targetcli/prefs.bin.
targetcli shell version 2.1.fb49
Copyright 2011-2013 by Datera, Inc and others.
For help on commands, type 'help'.

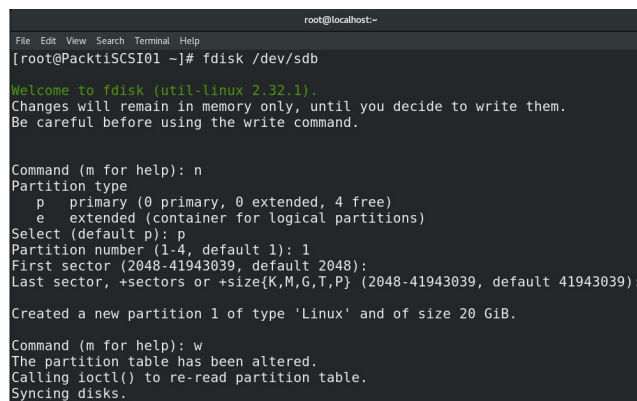
/> ls
0- / ..... [..]
  o- backstores ..... [..]
    o- block ..... [Storage Objects: 0]
    o- fileio ..... [Storage Objects: 0]
    o- pscsi ..... [Storage Objects: 0]
    o- ramdisk ..... [Storage Objects: 0]
    o- iscsi ..... [Targets: 0]
    o- loopback ..... [Targets: 0]

```

Figure 5.6 – The targetcli starting point – empty configuration

Let's start with the step-by-step procedure:

1. So, let's configure the XFS-based filesystem and configure the **LUN0** file image to be saved there. First, we need to partition the disk (in our case, **/dev/sdb**):



```

root@localhost~
[root@PacktiSCSI01 ~]# fdisk /dev/sdb

Welcome to fdisk (util-linux 2.32.1).
Changes will remain in memory only, until you decide to write them.
Be careful before using the write command.

Command (m for help): n
Partition type
   p   primary (0 primary, 0 extended, 4 free)
   e   extended (container for logical partitions)
Select (default p): p
Partition number (1-4, default 1): 1
First sector (2048-41943039, default 2048):
Last sector, +sectors or +size{K,M,G,T,P} (2048-41943039, default 41943039):

Created a new partition 1 of type 'Linux' and of size 20 GiB.

Command (m for help): w
The partition table has been altered.
Calling ioctl() to re-read partition table.
Syncing disks.

```

Figure 5.7 – Partitioning **/dev/sdb** for the XFS filesystem

2. The next step is to format this partition, create and use a directory called **/LUN0** to mount this

filesystem, and serve our **LUN0** image, which we're going to configure in the next steps:

```

root@localhost:~# mkfs.xfs /dev/sdb1 ; mkdir /LUN0 ; mount /dev/sdb1 /LUN0
meta-data=/dev/sdb1             isize=512    agcount=4, agsize=1310656 blks
                        =               sectsz=512   attr=2, projid32bit=1
                        =               crc=1        finobt=1, sparse=1, rmapbt=0
                        =               reflink=1
data              =               bsize=4096    blocks=5242624, imaxpct=25
                        =               sunit=0      swidth=0 blks
naming             =version 2          bsize=4096   ascii-ci=0, ftype=1
log                =internal log       bsize=4096   blocks=2560, version=2
                        =               sectsz=512   sunit=0 blks, lazy-count=1
realtime           =none               extsz=4096   blocks=0, rtextents=0

```

Figure 5.8 – Formatting the XFS filesystem, creating a directory, and mounting it to that directory

3. The next step is configuring **targetcli** so that it creates **LUN0** and assigns an image file for **LUN0**, which will be saved in the **/LUN0** directory. First, we need to start the **targetcli** command:

```

root@localhost:~# targetcli
targetcli shell version 2.1.fb40
Copyright 2011-2013 by Datera, Inc and others.
For help on commands, type 'help'.

/> /iscsi
/iscsi> create
Created target iqn.2003-01.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdbb11.
Created TPG 1.
Global pref auto_add_default_portal=true
Created default portal listening on all IPs (0.0.0.0), port 3260.
/iscsi> /backstores
/backstores> /backstores/fileio create LUN0 /LUN0/LUN0.img 20000M write_back=false
Created fileio LUN0 with size 20971520000
/backstores> ls
o- backstores ..... [Storage Objects: 0]
  o- block ..... [Storage Objects: 0]
  o- fileio ..... [Storage Objects: 1]
    o- LUN0 ..... [/LUN0/LUN0.img (19.5GiB) write-thru deactivated]
      o- alua ..... [ALUA Groups: 1]
        o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
  o- pscsi ..... [Storage Objects: 0]
  o- ramdisk ..... [Storage Objects: 0]

```

Figure 5.9 – Creating an iSCSI target, LUN0, and hosting it as a file

4. Next, let's configure a block device-based LUN backend—**LUN2**—which is going to use **/dev/sdc1** (create the partition using the previous example) and check the current state:

```

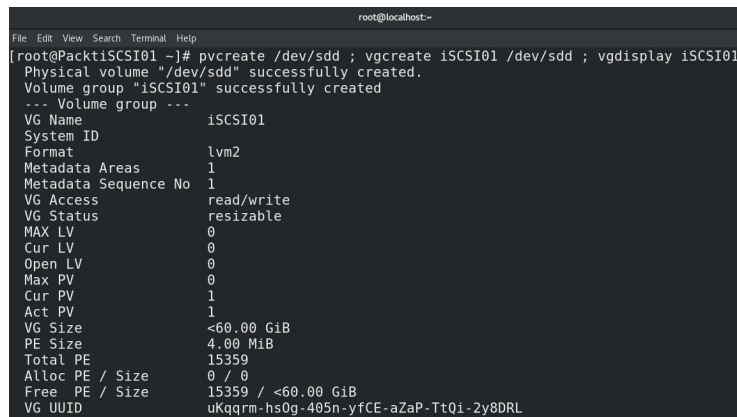
root@localhost:~# /backstores/block create name=LUN1 dev=/dev/sdc1
Created block storage object LUN1 using /dev/sdc1.
/backstores> cd /
/> ls
o- / ..... [Storage Objects: 1]
  o- backstores ..... [Storage Objects: 1]
    o- block ..... [Storage Objects: 1]
      o- LUN1 ..... [/dev/sdc1(40.0GiB) write-thru deactivated]
        o- alua ..... [ALUA Groups: 1]
          o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
    o- fileio ..... [Storage Objects: 1]
      o- LUN0 ..... [/LUN0/LUN0.img (19.5GiB) write-thru deactivated]
        o- alua ..... [ALUA Groups: 1]
          o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
    o- pscsi ..... [Storage Objects: 0]
    o- ramdisk ..... [Storage Objects: 0]
  o- iscsi ..... [Targets: 1]
    o- iqn.2003-01.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdbb11 ..... [TPGs: 1]
      o- tpg1 ..... [no-gen-acls, no-auth]
        o- acls ..... [ACLs: 0]
        o- luns ..... [LUNs: 0]
        o- portals ..... [Portals: 1]
          o- 0.0.0.0:3260 ..... [OK]
    o- loopback ..... [Targets: 0]

```


Figure 5.10 – Creating LUN1, hosting it directly from a block device

So, **LUN0** and **LUN1** and their respective backends are now configured. Let's finish things off by configuring LVM:

1. First, we are going to prepare the physical volume for LVM, create a volume group out of that volume, and display all the information about that volume group so that we can see how much space we have for **LUN2**:



```

root@localhost:~# pvcreate /dev/sdd ; vgcreate iSCSI01 /dev/sdd ; vgdisplay iSCSI01
Physical volume "/dev/sdd" successfully created.
Volume group "iSCSI01" successfully created
--- Volume group ---
VG Name                iSCSI01
System ID               lvm2
Format                  lvm2
Metadata Areas          1
Metadata Sequence No    1
VG Access                read/write
VG Status                resizable
MAX LV                  0
Cur LV                  0
Open LV                  0
Max PV                   0
Cur PV                  1
Act PV                   1
VG Size                  <60.00 GiB
PE Size                  4.00 MiB
Total PE                 15359
Alloc PE / Size          0 / 0
Free PE / Size           15359 / <60.00 GiB
VG UUID                  uKqqrn-hs0g-405n-yfCE-aZaP-TtQi-2y8DRL
  
```

Figure 5.11 – Configuring the physical volume for LVM, building a volume group, and displaying information about that volume group

2. The next step is to actually create the logical volume, which is going to be our block storage device backend for **LUN2** in the iSCSI target. We can see from the **vgdisplay** output that we have 15,359 4 MB blocks available, so let's use that to create our logical volume, called **LUN2**. Go to **targetcli** and configure the necessary settings for **LUN2**:

```

root@localhost:~# /backstores/block create name=LUN2 dev=/dev/iscsi01/LUN2
Created block storage object LUN2 using /dev/iscsi01/LUN2.
root@localhost:~# cd /
root@localhost:~# ls
.  ..  backstores
root@localhost:~# ls -lR backstores
backstores:
total 4
drwxr-xr-x. 2 root root 4096 Jul 29 08:17 block
drwxr-xr-x. 2 root root 4096 Jul 29 08:17 fileio
drwxr-xr-x. 2 root root 4096 Jul 29 08:17 pvc
drwxr-xr-x. 2 root root 4096 Jul 29 08:17 ramdisk
drwxr-xr-x. 2 root root 4096 Jul 29 08:17 iscsi
drwxr-xr-x. 2 root root 4096 Jul 29 08:17 loopback
backstores/block:
total 4
drwxr-xr-x. 2 root root 4096 Jul 29 08:17 LUN1
drwxr-xr-x. 2 root root 4096 Jul 29 08:17 LUN2
backstores/fileio:
total 4
drwxr-xr-x. 2 root root 4096 Jul 29 08:17 LUN0
backstores/iscsi:
total 4
drwxr-xr-x. 2 root root 4096 Jul 29 08:17 iqn.2003-01.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdbb11
backstores/loopback:
total 4
drwxr-xr-x. 2 root root 4096 Jul 29 08:17 0.0.0.0:3260

```

Figure 5.12 – Configuring LUN2 with the LVM backend

- Let's stop here for a second and switch to the KVM host (the iSCSI initiator) configuration. First, we need to install the iSCSI initiator, which is part of a package called **iscsi-initiator-utils**. So, let's use the **yum** command to install that:

```
yum -y install iscsi-initiator-utils
```

- Next, we need to configure the IQN of our initiator. We usually want this name to be reminiscent of the hostname, so, seeing that our host's FQDN is **PacktStratis01**, we'll use that to configure the IQN. To do that, we need to edit the **/etc/iscsi/initiatorname.iscsi** file and configure the **InitiatorName** option. For example, let's set it to **iqn.2019-12.com.packt:PacktStratis01**. The content of the **/etc/iscsi/initiatorname.iscsi** file should be as follows:

```
InitiatorName=iqn.2019-12.com.packt:PacktStratis01
```

- Now that this is configured, let's go back to the iSCSI target and create an **Access Control List (ACL)**. The ACL is going to allow our KVM

host's initiator to connect to the iSCSI target portal:

```

root@localhost:~# cd /iscsi/iqn.2003-01.org.linux-iscsi.packttscsi01.x8664:sn.7b3c2efdbb11/tpg1/
root@localhost:~# ls
o- tpg1 ..... [no-gen-acls, no-auth]
  o- acls ..... [ACLs: 0]
  o- luns ..... [LUNs: 0]
  o- portals ..... [Portals: 1]
    o- 0.0.0.0:3260 ..... [OK]
root@localhost:~# /iscsi/iqn.2003-01.org.linux-iscsi.packttscsi01.x8664:sn.7b3c2efdbb11/tpg1> cd acls
root@localhost:~# /iscsi/iqn.2003-01.org.linux-iscsi.packttscsi01.x8664:sn.7b3c2efdbb11/tpg1/acls> create iqn.2019-12.com.packt:packtstratis01
Created ACL for iqn.2019-12.com.packt:packtstratis01
root@localhost:~# /iscsi/iqn.2003-01.org.linux-iscsi.packttscsi01.x8664:sn.7b3c2efdbb11/tpg1/acls>

```

Figure 5.13 – Creating an ACL so that the KVM host's initiator can connect to the iSCSI target

- Next, we need to publish our pre-created file-based and block-based devices to the iSCSI target LUNs. So, we need to do this:

```

/iscsi/iqn.2003-01.org.linux-iscsi.packttscsi01.x8664:sn.7b3c2efdbb11/tpg1/acls> cd ../luns
/iscsi/iqn.2003-01.org.linux-iscsi.packttscsi01.x8664:sn.7b3c2efdbb11/tpg1/luns> create /backstores/fileio/LUN0
Created LUN 0
/iscsi/iqn.2003-01.org.linux-iscsi.packttscsi01.x8664:sn.7b3c2efdbb11/tpg1/luns> create /backstores/block/LUN1
Created LUN 1
/iscsi/iqn.2003-01.org.linux-iscsi.packttscsi01.x8664:sn.7b3c2efdbb11/tpg1/luns> create /backstores/block/LUN2
Created LUN 2
/iscsi/iqn.2003-01.org.linux-iscsi.packttscsi01.x8664:sn.7b3c2efdbb11/tpg1/luns>

```

Figure 5.14 – Adding our file-based and block-based devices to the iSCSI target LUNs 0, 1, and 2

The end result should look like this:

```

root@localhost:~# ls
o- backstores ..... [Storage Objects: 2]
  o- block ..... [Storage Objects: 2]
    o- LUN1 ..... [/dev/sdcl(40.0GiB) write-thru activated]
      o- alua ..... [ALUA Groups: 1]
        o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
    o- LUN2 ..... [/dev/iscsi01/LUN2 (60.0GiB) write-thru activated]
      o- alua ..... [ALUA Groups: 1]
        o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
  o- fileio ..... [Storage Objects: 1]
    o- LUN0 ..... [/LUN0/LUN0.img (19.5GiB) write-thru activated]
      o- alua ..... [ALUA Groups: 1]
        o- default_tg_pt_gp ..... [ALUA state: Active/optimized]
  o- pscsi ..... [Storage Objects: 0]
  o- ramdisk ..... [Storage Objects: 0]
  o- iscsi ..... [Targets: 1]
    o- iqn.2003-01.org.linux-iscsi.packttscsi01.x8664:sn.7b3c2efdbb11 ..... [TPGs: 1]
      o- tpg1 ..... [no-gen-acls, no-auth]
        o- acls ..... [ACLs: 1]
          o- iqn.2019-12.com.packt:packtstratis01 ..... [Mapped LUNs: 3]
            o- mapped_lun0 ..... [Lun0 fileio/LUN0 (rw)]
            o- mapped_lun1 ..... [Lun1 block/LUN1 (rw)]
            o- mapped_lun2 ..... [Lun2 block/LUN2 (rw)]
        o- luns ..... [LUNs: 3]
          o- lun0 ..... [fileio/LUN0 (/LUN0/LUN0.img) (default_tg_pt_gp)]
          o- lun1 ..... [block/LUN1 (/dev/sdcl) (default_tg_pt_gp)]
          o- lun2 ..... [block/LUN2 (/dev/iscsi01/LUN2) (default_tg_pt_gp)]
        o- portals ..... [Portals: 1]
          o- 0.0.0.0:3260 ..... [OK]
      o- loopback ..... [Targets: 0]

```

Figure 5.15 – The end result

At this point, everything is configured. We need to go back to our KVM host and define a storage pool that will use these LUNs. The easiest way to do that would be to use an XML configuration

file for the pool. So, here's our sample configuration XML file; we'll call it **iSCSIPool1.xml**:

```
<pool type='iscsi'>
  <name>MyiSCSIPool</name>
  <source>
    <host name='192.168.159.145' />
    <device path='iqn.2003-
01.org.linux-
iscsi.packtiscsi01.x8664:sn.7b3c2efdb
b11' />
  </source>
  <initiator>
    <iqn name='iqn.2019-
12.com.packt:PacktStratis01' />
  </initiator>
  <target>
    <path>/dev/disk/by-path</path>
  </target>
</pool>
```

Let's explain the file step by step:

- **pool type= 'iscsi'**: We're telling libvirt that this is an iSCSI pool.
- **name** : The pool name.
- **host name**: The IP address of the iSCSI target.
- **device path**: The IQN of the iSCSI target.
- The IQN name in the initiator section: The IQN of the initiator.
- **target path**: The location where iSCSI target's LUNs will be mounted.

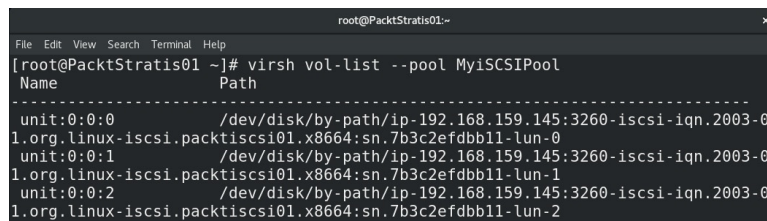
Now, all that's left for us to do is to define, start, and autostart our new iSCSI-backed KVM storage pool:

```
virsh pool-define --file
iSCSIPOOL.xml
virsh pool-start --pool MyiSCSIPOOL
virsh pool-autostart --pool
MyiSCSIPOOL
```

The target path part of the configuration can be easily checked via **virsh**. If we type the following command into the KVM host, we will get the list of available LUNs from the **MyiSCSIPOOL** pool that we just configured:

```
virsh vol-list --pool MyiSCSIPOOL
```

We get the following result for this command:



```
root@PacktStratis01:~# virsh vol-list --pool MyiSCSIPOOL
Name                                     Path
-----
unit:0:0:0                             /dev/disk/by-path/ip-192.168.159.145:3260-iscsi-iqn.2003-0
l.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdbb11-lun-0
unit:0:0:1                             /dev/disk/by-path/ip-192.168.159.145:3260-iscsi-iqn.2003-0
l.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdbb11-lun-1
unit:0:0:2                             /dev/disk/by-path/ip-192.168.159.145:3260-iscsi-iqn.2003-0
l.org.linux-iscsi.packtiscsi01.x8664:sn.7b3c2efdbb11-lun-2
```

Figure 5.16 – Runtime names for our iSCSI pool LUNs

If this output reminds you a bit of the VMware vSphere Hypervisor storage runtime names, you are definitely on the right track. We will be able to use these storage pools in [Chapter 7, Virtual Machine – Installation, Configuration, and Life-Cycle Management](#), when we start deploying our virtual machines.

Storage redundancy and multipathing

Redundancy is one of the keywords of IT, where any single component failure could mean big

problems for a company or its customers. The general design principle of avoiding SPOF is something that we should always stick to. At the end of the day, no network adapter, cable, switch, router, or storage controller is going to work forever. So, calculating redundancy into our designs helps our IT environment during its normal life cycle.

At the same time, redundancy can be combined with multipathing to also ensure higher throughput. For example, when we connect our physical host to FC storage with two controllers with four FC ports each, we can use four paths (if the storage is active-passive) or eight paths (if it's active-active) to the same LUN(s) exported from this storage device to a host. This gives us multiple additional options for LUN access, on top of the fact that it gives us more availability, even in the case of failure.

Getting a regular KVM host to do, for example, iSCSI multipathing is quite a bit complex. There are multiple configuration issues and blank spots in terms of documentation, and supportability of such a configuration is questionable. However, there are products that use KVM that support it out of the box, such as oVirt (which we covered before) and **Red Hat Enterprise Virtualization Hypervisor (RHEV-H)**. So, let's use oVirt for this example on iSCSI.

Before you do this, make sure that you have done the following:

- Your Hypervisor host is added to the oVirt inventory.
- Your Hypervisor host has two additional network cards, independent of the management network.
- The iSCSI storage has two additional network cards in the same L2 networks as the two additional hypervisor network cards.
- The iSCSI storage is configured so that it has at least a target and a LUN already configured in a way that will enable the hypervisor host to connect to it.

So, as we're doing this in oVirt, there are a couple of things that we need to do. First, from a networking perspective, it would be a good idea to create some storage networks. In our case, we're going to assign two networks for iSCSI, and we will call them **iSCSI01** and **iSCSI02**. We need to open the oVirt administration panel, hover over **Network**, and select **Networks** from the menu. This will open a pop-up window for the **New Logical Network** wizard. So, we just need to name the network **iSCSI01** (for the first one), uncheck the **VM network** checkbox (as this isn't a virtual machine network), and go to the **Cluster** tab, where we deselect the **Require all** checkbox. Repeat the whole process again for the **iSCSI02** network:

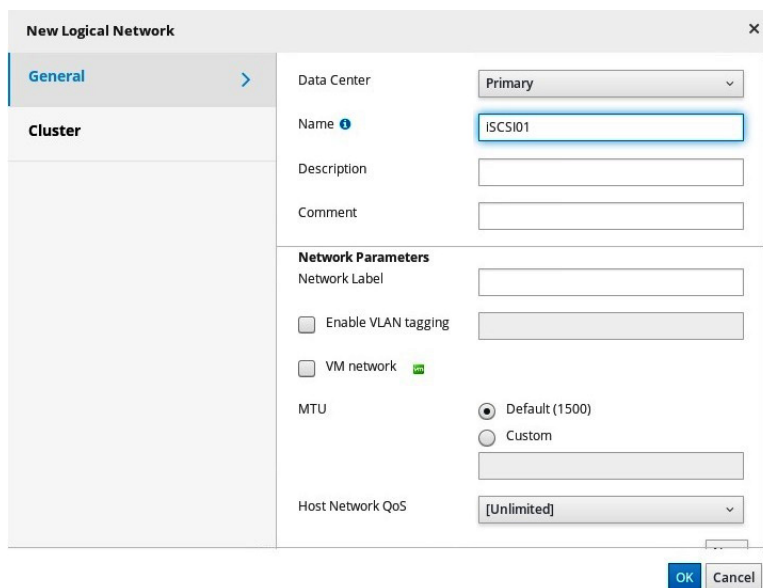


Figure 5.17 – Configuring networks for iSCSI bond

The next step is assigning these networks to host network adapters. Go to **compute/hosts**, double-click on the host that you added to oVirt's inventory, select the **Network interfaces** tab, and click on the **Setup Host Networks** icon in the top-right corner. In that UI, drag and drop **iSCSI01** on the second network interface and **iSCSI02** on the third network interface. The first network interface is already taken by the oVirt management network. It should look something like this:

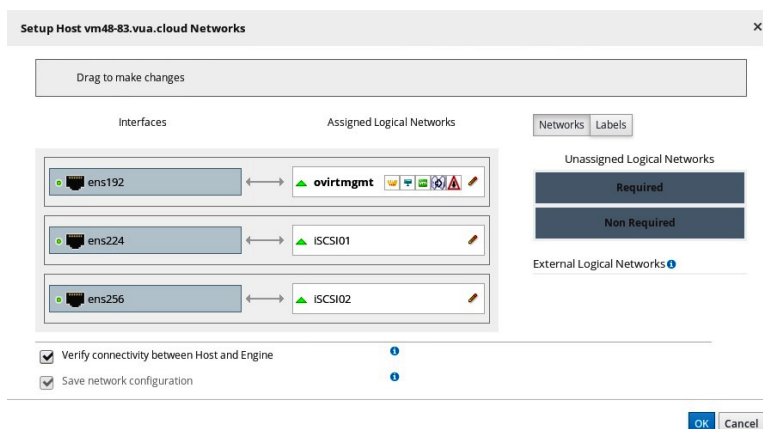


Figure 5.18 – Assigning virtual networks to the hypervisor's physical adapters

Before you close the window down, make sure that you click on the *pencil* sign on both **iSCSI01** and **iSCSI02** to set up IP addresses for these two virtual networks. Assign network configuration that can connect you to your iSCSI storage on the same or different subnets:

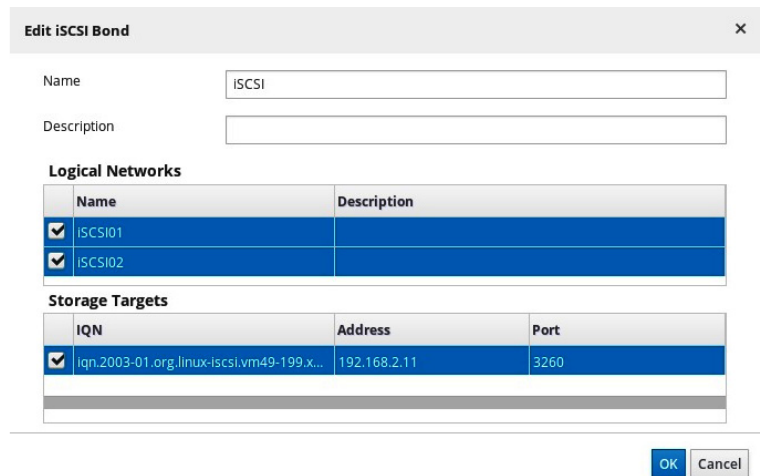


Figure 5.19 – Creating an iSCSI bond on the data center level

There you go, you have just configured an iSCSI bond. The last part of our configuration is enabling it. Again, in the oVirt GUI, go to **Compute | Data Centers**, select your datacenter with a double-click, and go to the **iSCSI Multipathing** tab:

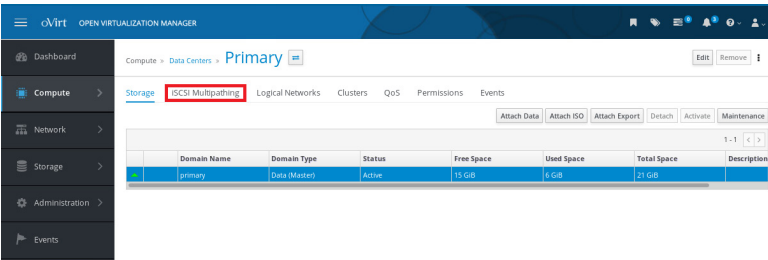


Figure 5.20 – Configuring iSCSI multipathing on the data center level

Click on the **Add** button at the top-right side and go through the wizard. Specifically, select both

the **iSCSI01** and **iSCSI02** networks in the top part of the pop-up window, and the iSCSI target on the lower side.

Now that we have covered the basics of storage pools, NFS, and iSCSI, we can move on to a standard open source way of deploying storage infrastructure, which would be to use Gluster and/or Ceph.

Gluster and Ceph as a storage backend for KVM

There are other advanced types of filesystems that can be used as the libvirt storage backend. So, let's now discuss two of them—Gluster and Ceph. Later, we'll also check how libvirt works with GFS2.

Gluster

Gluster is a distributed filesystem that's often used for high-availability scenarios. Its main advantages over other filesystems are the fact that it's scalable, it can use replication and snapshots, it can work on any server, and it's usable as a basis for shared storage—for example, via NFS and SMB. It was developed by a company called Gluster Inc., which was acquired by RedHat in 2011. However, unlike Ceph, it's a *file* storage service, while Ceph offers *block* and *object*-based storage. Object-based storage for block-based devices means direct, binary storage, directly to a LUN. There are no filesystems involved, which

theoretically means less overhead as there's no filesystem, filesystem tables, and other constructs that might slow the I/O process down.

Let's first configure Gluster to show its use case with libvirt. In production, that means installing at least three Gluster servers so that we can make high availability possible. Gluster configuration is really straightforward, and in our example, we are going to create three CentOS 7 machines that we will use to host the Gluster filesystem. Then, we will mount that filesystem on our hypervisor host and use it as a local directory. We can use GlusterFS directly from libvirt, but the implementation is just not as refined as using it via the gluster client service, mounting it as a local directory, and using it directly as a directory pool in libvirt.

Our configuration will look like this:

Hostname	gluster1	gluster2	gluster3
IP	192.168.159.147/24	192.168.159.148/24	192.168.159.149/24
Gluster disk	/dev/sdb (10GB)	/dev/sdb (10GB)	/dev/sdb (10GB)
Gluster local directory	/gluster/bricks/1	/gluster/bricks/1	/gluster/bricks/1

Figure 5.21 – Basic settings for our Gluster cluster

So, let's put that into production. We have to issue a large sequence of commands on all of the servers before we configure Gluster and expose it to our KVM host. Let's start with **gluster1**. First, we are going to do a system-wide update and reboot to prepare the core operating system for Gluster installation. Type the following commands into all three CentOS 7 servers:

```
yum -y install epel-release*
yum -y install centos-release-
gluster7.noarch
yum -y update
yum -y install glusterfs-server
systemctl reboot
```

Then, we can start deploying the necessary repositories and packages, format disks, configure the firewall, and so on. Type the following commands into all the servers:

```
mkfs.xfs /dev/sdb
mkdir /gluster/bricks/1 -p
echo '/dev/sdb /gluster/bricks/1 xfs
defaults 0 0' >> /etc/fstab
mount -a
mkdir /gluster/bricks/1/brick
systemctl disable firewalld
systemctl stop firewalld
systemctl start glusterd
systemctl enable glusterd
```

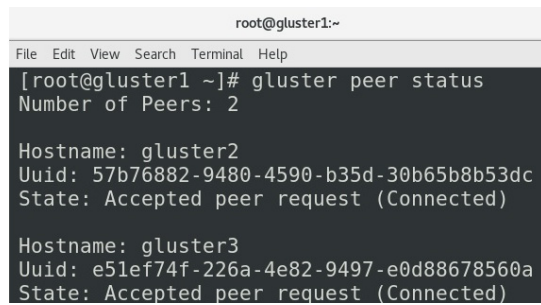
We need to do a bit of networking configuration as well. It would be good if these three servers can *resolve* each other, which means either configuring a DNS server or adding a couple of lines to our **/etc/hosts** file. Let's do the latter. Add the following lines to your **/etc/hosts** file:

```
192.168.159.147 gluster1
192.168.159.148 gluster2
192.168.159.149 gluster3
```

For the next part of the configuration, we can just log in to the first server and use it as the de facto management server for our Gluster infrastructure. Type in the following commands:

```
gluster peer probe gluster1
gluster peer probe gluster2
gluster peer probe gluster3
gluster peer status
```

The first three commands should get you the **peer probe: success** status. The third one should return an output similar to this:



```
root@gluster1:~
File Edit View Search Terminal Help
[root@gluster1 ~]# gluster peer status
Number of Peers: 2

Hostname: gluster2
Uuid: 57b76882-9480-4590-b35d-30b65b8b53dc
State: Accepted peer request (Connected)

Hostname: gluster3
Uuid: e51ef74f-226a-4e82-9497-e0d88678560a
State: Accepted peer request (Connected)
```

Figure 5.22 – Confirmation that the Gluster servers peered successfully

Now that this part of the configuration is done, we can create a Gluster-distributed filesystem. We can do this by typing the following sequence of commands:

```
gluster volume create kvmgluster
replica 3 \
gluster1:/gluster/bricks/1/brick
gluster2:/gluster/bricks/1/brick \
gluster3:/gluster/bricks/1/brick
gluster volume start kvmgluster
gluster volume set kvmgluster
auth.allow 192.168.159.0/24
gluster volume set kvmgluster allow-
insecure on
gluster volume set kvmgluster
storage.owner-uid 107
gluster volume set kvmgluster
storage.owner-gid 107
```

Then, we could mount Gluster as an NFS directory for testing purposes. For example, we can create a distributed namespace called **kvmgluster** to all of the member hosts (**gluster1**, **gluster2**, and **gluster3**). We can do this by using the following commands:

```
echo 'localhost:/kvmgluster /mnt
glusterfs \
defaults,_netdev,backupvolfile-
server=localhost 0 0' >> /etc/fstab
mount.glusterfs localhost:/kvmgluster
/mnt
```

The Gluster part is now ready, so we need to go back to our KVM host and mount the Gluster filesystem to it by typing in the following commands:

```
wget \
https://download.gluster.org/pub/gluster/glusterfs/6/LATEST/CentOS/gl\
usterfs-rhel8.repo -P
/etc/yum.repos.d
yum install glusterfs glusterfs-fuse
attr -y
mount -t glusterfs -o
context="system_u:object_r:virt_image
_t:s0" \ gluster1:/kvmgluster
/var/lib/libvirt/images/GlusterFS
```

We have to pay close attention to Gluster releases on the server and client, which is why we downloaded the Gluster repository information for CentOS 8 (we're using it on the KVM server) and installed the necessary Gluster client pack-

ages. That enabled us to mount the filesystem with the last command.

Now that we've finished our configuration, we just need to add this directory as a libvirt storage pool. Let's do that by using an XML file with the storage pool definition, which contains the following entries:

```
<pool type='dir'>
  <name>glusterfs-pool</name>
  <target>
    <path>/var/lib/libvirt/images/Glu
sterFS</path>
    <permissions>
      <mode>0755</mode>
      <owner>107</owner>
      <group>107</group>
      <label>system_u:object_r:virt_i
mage_t:s0</label>
    </permissions>
  </target>
</pool>
```

Let's say that we saved this file in the current directory, and that the file is called **gluster.xml**. We can import and start it in libvirt by using the following **virsh** commands:

```
virsh pool-define --file gluster.xml
virsh pool-start --pool glusterfs-
pool
virsh pool-autostart --pool
glusterfs-pool
```

We should mount this pool automatically on boot so that libvirt can use it. Therefore, we need to

add the following line to **/etc/fstab**:

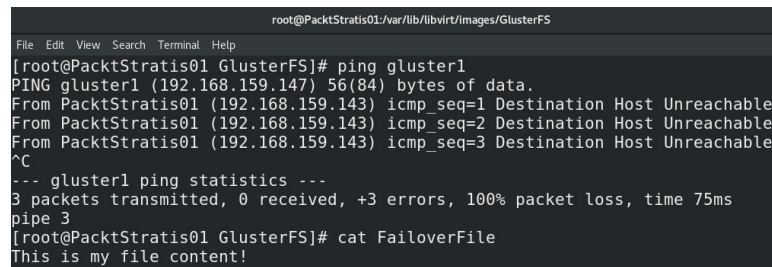
```
gluster1:/kvmgluster      /var/lib/libvirt/images/GlusterFS \
glusterfs defaults,_netdev 0 0
```

Using a directory-based approach enables us to avoid two problems that libvirt (and its GUI interface, **virt-manager**) has with Gluster storage pools:

- We can use Gluster's failover capability, which will be managed automatically by the Gluster utilities that we installed directly, as libvirt doesn't support them yet.
- We will avoid creating virtual machine disks *manually*, which is another limitation of libvirt's implementation of Gluster support, while directory-based storage pools support it without any issues.

It seems weird that we're mentioning *failover*, as it seems as though we didn't configure it as a part of any of the previous steps. Actually, we have. When we issued the last mount command, we used Gluster's built-in modules to establish connectivity to the *first* Gluster server. That, in turn, means that after this connection, we got all of the details about the whole Gluster pool, which we configured so that it's hosted on three servers. If any kind of failure happens—which we can easily simulate—this connection will continue working. We can simulate this scenario by turning off any of the Gluster servers, for example—**gluster1**. You'll see that the local directory where we mounted Gluster directory still works,

even though **gluster1** is down. Let's see that in action (the default timeout period is 42 seconds):



```

root@PacktStratis01:/var/lib/libvirt/images/GlusterFS
File Edit View Search Terminal Help
[root@PacktStratis01 GlusterFS]# ping gluster1
PING gluster1 (192.168.159.147) 56(84) bytes of data.
From PacktStratis01 (192.168.159.143) icmp_seq=1 Destination Host Unreachable
From PacktStratis01 (192.168.159.143) icmp_seq=2 Destination Host Unreachable
From PacktStratis01 (192.168.159.143) icmp_seq=3 Destination Host Unreachable
^C
--- gluster1 ping statistics ---
3 packets transmitted, 0 received, +3 errors, 100% packet loss, time 75ms
pipe 3
[root@PacktStratis01 GlusterFS]# cat FailoverFile
This is my file content!

```

Figure 5.23 – Gluster failover working; the first node is down, but we're still able to get our files

If we want to be more aggressive, we can shorten this timeout period to—for example—2 seconds by issuing the following command on any of our Gluster servers:

```
gluster volume set kvmgcluster
network.ping-timeout number
```

The **number** part is in seconds, and by assigning it a lower number, we can directly influence how aggressive the failover process is.

So, now that everything is configured, we can start using the Gluster pool to deploy virtual machines, which we will discuss further in [*Chapter 7, Virtual Machine – Installation, Configuration, and Life-Cycle Management*](#).

Seeing as Gluster is a file-based backend that can be used for libvirt, it's only natural to describe how to use an advanced block-level and object-level storage backend. That's where Ceph comes in, so let's work on that now.

Ceph

Ceph can act as file-, block-, and object-based storage. But for the most part, we're usually using it as either block- or object-based storage. Again, this is a piece of open source software that's designed to work on any server (or a virtual machine). In its core, Ceph runs an algorithm called **Controlled Replication Under Scalable Hashing (CRUSH)**. This algorithm tries to distribute data across object devices in a pseudo-random manner, and in Ceph, it's managed by a cluster map (a CRUSH map). We can easily scale Ceph out by adding more nodes, which will redistribute data in a minimum fashion to ensure as small amount of replication as possible.

An internal Ceph component called **Reliable Autonomic Distributed Object Store (RADOS)** is used for snapshots, replication, and thin provisioning. It's an open source project that was developed by the University of California.

Architecture-wise, Ceph has three main services:

- **ceph-mon** : Used for cluster monitoring, CRUSH maps, and **Object Storage Daemon (OSD)** maps.
- **ceph-osd**: This handles actual data storage, replication, and recovery. It requires at least two nodes; we'll use three for clustering reasons.
- **ceph-mds**: Metadata server, used when Ceph needs filesystem access.

In accordance with best practices, make sure that you always design your Ceph environments with the key principles in mind—all of the data nodes need to have the same configuration. That means the same amount of memory, the same storage controllers (don't use RAID controllers, just plain HBAs without RAID firmware if possible), the same disks, and so on. That's the only way to ensure a constant level of Ceph performance in your environments.

One very important aspect of Ceph is data placement and how placement groups work. Placement groups offer us a chance to split the objects that we create and place them in OSDs in an optimal fashion. Translation: the bigger the number of placement groups we configure, the better balance we're going to get.

So, let's configure Ceph from scratch. We're going to follow the best practices again and deploy Ceph by using five servers—one for administration, one for monitoring, and three OSDs.

Our configuration will look like this:

hostname	ceph-admin	ceph-monitor	ceph-osd1	ceph-osd2	ceph-osd3
IP/24	192.168.159.150	192.168.159.151	192.168.159.152	192.168.159.153	192.168.159.154
Ceph disk	/dev/sdb	/dev/sdb	/dev/sdb	/dev/sdb	/dev/sdb

Figure 5.24 – Basic Ceph configuration for our infrastructure

Make sure that these hosts can resolve each other via DNS or `/etc/hosts`, and that you configure all of them to use the same NTP source. Make sure that you update all of the hosts by using the following:

```
yum -y update; reboot
```

Also, make sure that you type the following commands into all of the hosts as the *root* user. Let's start by deploying packages, creating an admin user, and giving them rights to **sudo**:

```
rpm -Uhv
http://download.ceph.com/rpm-
jewel/el7/noarch/ceph-release-1-
1.el7.noarch.rpm
yum -y install ceph-deploy ceph ceph-
radosgw
useradd cephadmin
echo "cephadmin:ceph123" | chpasswd
echo "cephadmin ALL = (root)
NOPASSWD:ALL" | sudo tee
/etc/sudoers.d/cephadmin
chmod 0440 /etc/sudoers.d/cephadmin
```

Disabling SELinux will make our life easier for this demonstration, as will getting rid of the firewall:

```
sed -i
's/SELINUX=enforcing/SELINUX=disabled
/g' /etc/selinux/config
systemctl stop firewalld
systemctl disable firewalld
systemctl mask firewalld
```

Let's add hostnames to **/etc/hosts** so that administration is easier for us:

```
echo "192.168.159.150 ceph-admin" >>
/etc/hosts
echo "192.168.159.151 ceph-monitor"
>> /etc/hosts
```

```
echo "192.168.159.152 ceph-osd1" >>  
/etc/hosts  
echo "192.168.159.153 ceph-osd2" >>  
/etc/hosts  
echo "192.168.159.154 ceph-osd3" >>  
/etc/hosts
```

Change the last **echo** part to suit your environment—hostnames and IP addresses. We're just using this as an example from our environment. The next step is making sure that we can use our admin host to connect to all of the hosts. The easiest way to do that is by using SSH keys. So, on **ceph-admin**, log in as root and type in the **ssh-keygen** command, and then press the *Enter* key all the way through. It should look something like this:



Figure 5.25 – Generating an SSH key for root for Ceph setup purposes

We also need to copy this key to all of the hosts. So, again, on **ceph-admin**, use **ssh-copy-id** to copy the keys to all of the hosts:

```
ssh-copy-id cephadmin@ceph-admin  
ssh-copy-id cephadmin@ceph-monitor
```

```
ssh-copy-id cephadmin@ceph-osd1
ssh-copy-id cephadmin@ceph-osd2
ssh-copy-id cephadmin@ceph-osd3
```

Accept all of the keys when SSH asks you, and use **ceph123** as the password, which we selected in one of the earlier steps. After all of this is done, there's one last step that we need to do on **ceph-admin** before we start deploying Ceph—we have to configure SSH to use the **cephadmin** user as a default user to log in to all of the hosts. We will do this by going to the **.ssh** directory as root on **ceph-admin**, and creating a file called **config** with the following content:

```
Host ceph-admin
    Hostname ceph-admin
    User cephadmin
Host ceph-monitor
    Hostname ceph-monitor
    User cephadmin
Host ceph-osd1
    Hostname ceph-osd1
    User cephadmin
Host ceph-osd2
    Hostname ceph-osd2
    User cephadmin
Host ceph-osd3
    Hostname ceph-osd3
    User cephadmin
```

That was a long pre-configuration, wasn't it? Now it's time to actually start deploying Ceph. The first step is to configure **ceph-monitor**. So, on **ceph-admin**, type in the following commands:

```
cd /root
mkdir cluster
cd cluster
ceph-deploy new ceph-monitor
```

Because of the fact that we selected a configuration in which we have three OSDs, we need to configure Ceph so that it uses these additional two hosts. So, in the **cluster** directory, edit the file called **ceph.conf** and add the following two lines at the end:

```
public network = 192.168.159.0/24
osd pool default size = 2
```

This will make sure that we can only use our example network (**192.168.159.0/24**) for Ceph, and that we have two additional OSDs on top of the original one.

Now that everything's ready, we have to issue a sequence of commands to configure Ceph. So, again, on **ceph-admin**, type in the following commands:

```
ceph-deploy install ceph-admin ceph-
monitor ceph-osd1 ceph-osd2 ceph-osd3
ceph-deploy mon create-initial
ceph-deploy gatherkeys ceph-monitor
ceph-deploy disk list ceph-osd1 ceph-
osd2 ceph-osd3
ceph-deploy disk zap ceph-
osd1:/dev/sdb ceph-
osd2:/dev/sdb ceph-osd3:/dev/sdb
ceph-deploy osd prepare ceph-
osd1:/dev/sdb ceph-osd2:/dev/sdb
ceph-osd3:/dev/sdb
```

```
ceph-deploy osd activate ceph-  
osd1:/dev/sdb1 ceph-osd2:/dev/sdb1  
ceph-osd3:/dev/sdb1
```

Let's describe these commands one by one:

- The first command starts the actual deployment process—for the admin, monitor, and OSD nodes, with the installation of all the necessary packages.
- The second and third commands configure the monitor host so that it's ready to accept external connections.
- The two disk commands are all about disk preparation—Ceph will clear the disks that we assigned to it (`/dev/sdb` per OSD host) and create two partitions on them, one for Ceph data and one for the Ceph journal.
- The last two commands prepare these filesystems for use and activate Ceph. If at any time your **ceph-deploy** script stops, check your DNS and `/etc/hosts` and `firewalld` configuration, as that's where the problems usually are.

We need to expose Ceph to our KVM host, which means that we have to do a bit of extra configuration. We're going to expose Ceph as an object pool to our KVM host, so we need to create a pool. Let's call it **KVMpool1**. Connect to **ceph-admin**, and issue the following commands:

```
ceph osd pool create KVMpool 128 128
```

This command will create a pool called **KVMpool1**, with 128 placement groups.

The next step involves approaching Ceph from a security perspective. We don't want anyone connecting to this pool, so we're going to create a key for authentication to Ceph, which we're going to use on the KVM host for authentication purposes. We do that by typing the following command:

```
ceph auth get-or-create
client.KVMpool mon 'allow r' osd
'allow rwx pool=KVMpool'
```

It's going to throw us a status message, something like this:

```
key =
AQB9p8RdqS09CBAA1DHsiZJbehb7ZBffhfmFJ
Q==
```

We can then switch to the KVM host, where we need to do two things:

- Define a secret—an object that's going to link libvirt to a Ceph user—and by doing that, we're going to create a secret object with its **Universally Unique Identifier (UUID)**.
- Use that secret's UUID to link it to the Ceph key when we define the Ceph storage pool.

The easiest way to do these two steps would be by using two XML configuration files for libvirt. So, let's create those two files. Let's call the first one, **secret.xml**, and here are its contents:

```
<secret ephemeral='no'
private='no'>
  <usage type='ceph'>
```

```
<name>client.KVMpool
secret</name>
</usage>
</secret>
```

Make sure that you save and import this XML file by typing in the following command:

```
virsh secret-define --file secret.xml
```

After you press the *Enter* key, this command is going to throw out a UUID. Please copy and paste that UUID someplace safe, as we're going to need it for the pool XML file. In our environment, this first **virsh** command threw out the following output:

```
Secret 95b1ed29-16aa-4e95-9917-
c2cd4f3b2791 created
```

We need to assign a value to this secret so that when libvirt tries to use this secret, it knows which *password* to use. That's actually the password that we created on the Ceph level, when we used **ceph auth get-create**, which threw us the key. So, now that we have both the secret UUID and the Ceph key, we can combine them to create a complete authentication object. On the KVM host, we need to type in the following command:

```
virsh secret-set-value 95b1ed29-16aa-
4e95-9917-c2cd4f3b2791
AQB9p8RdqS09CBAA1DHsiZJbehb7ZBffhfmFJ
Q==
```

Now, we can create the Ceph pool file. Let's call the config file **ceph.xml**, and here are its contents:

```

<pool type="rbd">
  <source>
    <name>KVMpool</name>
    <host name='192.168.159.151'
port='6789' />
    <auth username='KVMpool'
type='ceph'>
      <secret uuid='95b1ed29-16aa-
4e95-9917-c2cd4f3b2791' />
    </auth>
  </source>
</pool>

```

So, the UUID from the previous step was used in this file to reference which secret (identity) is going to be used for Ceph pool access. Now we need to do the standard procedure—import the pool, start it, and autostart it—if we want to use it permanently (after the KVM host reboot). So, let's do that with the following sequence of commands on the KVM host:

```

virsh pool-define --file ceph.xml
virsh pool-start KVMpool
virsh pool-autostart KVMpool
virsh pool-list --details

```

The last command should produce an output similar to this:

```

[root@PacktStratis01 ~]# virsh pool-list --details

```

Name	State	Autostart	Persistent	Capacity	Allocation	Available
default	running	yes	yes	12.49 GiB	4.26 GiB	8.22 GiB
glusterfs-pool	running	yes	yes	9.99 GiB	134.61 MiB	9.86 GiB
KVMpool	running	yes	yes	14.97 GiB	0.00 B	14.65 GiB
MyISCSIPool	running	yes	yes	119.53 GiB	119.53 GiB	0.00 B
MyNFSPool	running	yes	yes	12.49 GiB	4.13 GiB	8.35 GiB

Figure 5.26 – Checking the state of our pools; the Ceph pool is configured and ready to be used

Now that the Ceph object pool is available for our KVM host, we could install a virtual machine on it. We're going to work on that – again – in [*Chapter 7, Virtual Machine – Installation, Configuration, and Life-Cycle Management*](#).

Virtual disk images and formats and basic KVM storage operations

Disk images are standard files stored on the host's filesystem. They are large and act as virtualized hard drives for guests. You can create such files using the **dd** command, as shown:

```
# dd if=/dev/zero  
of=/vms/dbvm_disk2.img bs=1G count=10
```

Here is the translation of this command for you:

Duplicate data (**dd**) from the input file (**if**) of **/dev/zero** (virtually limitless supply of zeros) into the output file (**of**) of **/vms/dbvm_disk2.img** (disk image) using blocks of 1 G size (**bs** = block size) and repeat this (**count**) just once (**10**).

Important note:

***dd** is known to be a resource-hungry command. It may cause I/O problems on the host system, so it's good to first check the available free memory and I/O state of the host system, and only then run it. If the system is already loaded, lower the block size to MB and increase the count to match the size of*

*the file you wanted (use **bs=1M**, **count=10000** instead of **bs=1G**, **count=10**).*

`/vms/dbvm_disk2.img` is the result of the preceding command. The image now has 10 GB preallocated and ready to use with guests either as the boot disk or second disk. Similarly, you can also create thin-provisioned disk images. Preallocated and thin-provisioned (sparse) are disk allocation methods, or you may also call it the format:

- **Preallocated:** A preallocated virtual disk allocates the space right away at the time of creation. This usually means faster write speeds than a thin-provisioned virtual disk.
- **Thin-provisioned:** In this method, space will be allocated for the volume as needed—for example, if you create a 10 GB virtual disk (disk image) with sparse allocation. Initially, it would just take a couple of MB of space from your storage and grow as it receives write from the virtual machine up to 10 GB size. This allows storage over-commitment, which means *faking* the available capacity from a storage perspective. Furthermore, this can lead to problems later, when storage space gets filled. To create a thin-provisioned disk, use the **seek** option with the **dd** command, as shown in the following command:

```
dd if=/dev/zero  
of=/vms/dbvm_disk2_seek.img bs=1G  
seek=10 count=0
```

Each comes with its own advantages and disadvantages. If you are looking for I/O performance,

go for a preallocated format, but if you have a non-IO-intensive load, choose thin-provisioned.

Now, you might be wondering how you can identify what disk allocation method a certain virtual disk uses. There is a good utility for finding this out: **qemu-img**. This command allows you to read the metadata of a virtual image. It also supports creating a new disk and performing low-level format conversion.

Getting image information

The **info** parameter of the **qemu-img** command displays information about a disk image, including the absolute path of the image, the file format, and the virtual and disk size. By looking at the virtual disk size from a QEMU perspective and comparing that to the image file size on the disk, you can easily identify what disk allocation policy is in use. As an example, let's look at two of the disk images we created:

```
# qemu-img info /vms/dbvm_disk2.img
image: /vms/dbvm_disk2.img
file format: raw
virtual size: 10G (10737418240 bytes)
disk size: 10G
# qemu-img info
/vms/dbvm_disk2_seek.img
image: /vms/dbvm_disk2_seek.img
file format: raw
virtual size: 10G (10737418240 bytes)
disk size: 10M
```

See the **disk size** line of both the disks. It's showing **10G** for `/vms/dbvm_disk2.img`, whereas for `/vms/dbvm_disk2_seek.img`, it's showing **10M** MiB. This difference is because the second disk uses a thin-provisioning format. The virtual size is what guests see and the disk size is what space the disk reserved on the host. If both the sizes are the same, it means the disk is preallocated. A difference means that the disk uses the thin-provisioning format. Now, let's attach the disk image to a virtual machine; you can attach it using **virt-manager** or the CLI alternative, **virsh**.

Attaching a disk using virt-manager

Start virt-manager from the host system's graphical desktop environment. It can also be started remotely using SSH, as demonstrated in the following command:

```
ssh -X host's address  
[remotehost]# virt-manager
```

So, let's use the Virtual Machine Manager to attach the disk to the virtual machine:

1. In the Virtual Machine Manager main window, select the virtual machine to which you want to add the secondary disk.
2. Go to the virtual hardware details window and click on the **Add Hardware** button located at the bottom-left side of the dialog box.
3. In **Add New Virtual Hardware**, select **Storage** and select the **Create a disk image for the virtual machine** button and virtual disk size, as in the following screenshot:

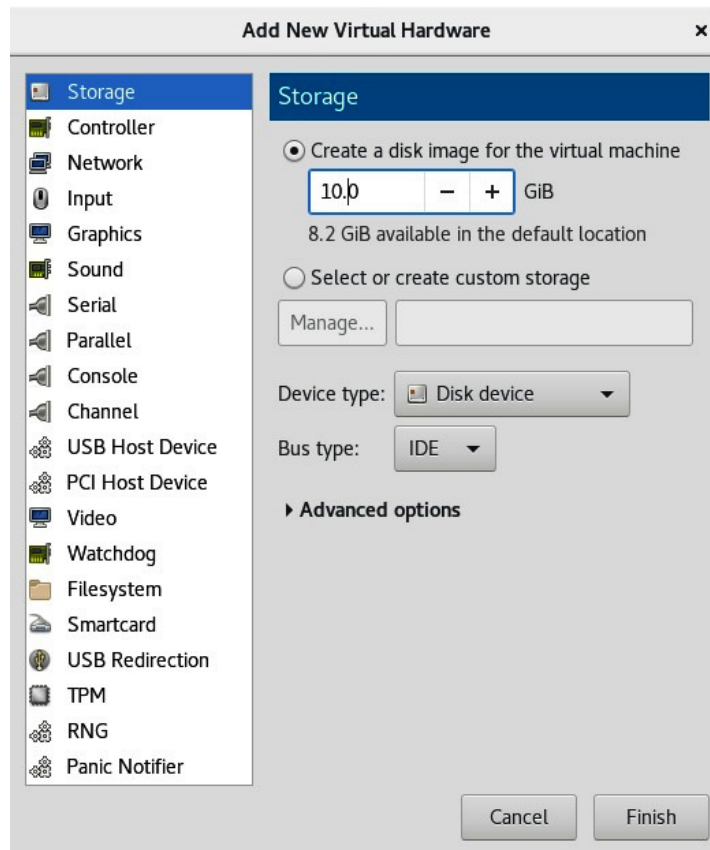


Figure 5.27 – Adding a virtual disk in virt-manager

4. If you want to attach the previously created **dbvm_disk2.img** image, choose **Select** or create custom storage, click on **Manage**, and either browse and point to the **dbvm_disk2.img** file from the **/vms** directory or find it in the local storage pool, then select it and click **Finish**.

Important note:

*Here, we used a disk image, but you are free to use any storage device that is present on the host system, such as a LUN, an entire physical disk (**/dev/sdb**) or disk partition (**/dev/sdb1**), or LVM logical volume. We could have used any of the previously configured storage pools for storing this image either as a file or object or directly to a block device.*

5. Clicking on the **Finish** button will attach the selected disk image (file) as a second disk to

the virtual machine using the default configuration. The same operation can be quickly performed using the **virsh** command.

Using virt-manager to create a virtual disk was easy enough—just a couple of clicks of a mouse and a bit of typing. Now, let's see how we can do that via the command line—namely, by using **virsh**.

Attaching a disk using virsh

virsh is a very powerful command-line alternative to virt-manager. You can perform an action in a second that would take minutes to perform through a graphical interface such as virt-manager. It provides an **attach-disk** option to attach a new disk device to a virtual machine. There are lots of switches provided with **attach-disk**:

```
attach-disk domain source target [[-  
-live] [--config] | [--current]] | [  
-persistent]] [--targetbusbus] [--  
driver driver] [--subdriversubdriver]  
[--iothreadiothread] [--cache cache]  
[--type type] [--mode mode] [--  
sourcetypesourcetype] [--serial  
serial] [--wwnwn] [--rawio] [--  
address address] [--multifunction] [  
-print-xml]
```

However, in a normal scenario, the following are sufficient to perform hot-add disk attachment to a virtual machine:

```
# virsh attach-disk CentOS8  
/vms/dbvm_disk2.img vdb --live --
```

config

Here, **CentOS8** is the virtual machine to which a disk attachment is executed. Then, there is the path of the disk image. **vdb** is the target disk name that would be visible inside the guest operating system. **--live** means performing the action while the virtual machine is running, and **--config** means attaching it persistently across reboot. Not adding a **--config** switch will keep the disk attached only until reboot.

Important note:

*Hot plugging support: The **acpihp** kernel module should be loaded in a Linux guest operating system in order to recognize a hot-added disk; **acpihp** provides legacy hot plugging support, whereas **pciehp** provides native hot plugging support. **pciehp** is dependent on **acpihp**. Loading **acpihp** will automatically load **pciehp** as a dependency.*

You can use the **virsh domblklist <vm_name>** command to quickly identify how many vDisks are attached to a virtual machine. Here is an example:

```
# virsh domblklist CentOS8 --details
Type Device Target Source
-----
-----
file disk vda
/var/lib/libvirt/images/fedora21.qcow
2
```

```
file disk vdb  
/vms/dbvm_disk2_seek.img
```

This clearly indicates that the two vDisks connected to the virtual machine are both file images. They are visible to the guest operating system as **vda** and **vdb**, respectively, and in the last column of the disk images path on the host system.

Next, we are going to see how to create an ISO library.

Creating an ISO image library

Although a guest operating system on the virtual machine can be installed from physical media by carrying out a passthrough the host's CD/DVD drive to the virtual machine, it's not the most efficient way. Reading from a DVD drive is slow compared to reading ISO from a hard disk, so a better way is to store ISO files (or logical CDs) used to install operating systems and applications for the virtual machines in a file-based storage pool and create an ISO image library.

To create an ISO image library, you can either use `virt-manager` or a **virsh** command. Let's see how to create an ISO image library using the **virsh** command:

1. First, create a directory on the host system to store the **.iso** images:

```
# mkdir /iso
```

2. Set the correct permissions. It should be owned by a root user with permission set to

700. If SELinux is in enforcing mode, the following context needs to be set:

```
# chmod 700 /iso
# semanage fcontext -a -t
virt_image_t "/iso(/.*)?"
```

3. Define the ISO image library using the **virsh** command, as shown in the following code block:

```
# virsh pool-define-as iso_library
dir - - - - "/iso"
# virsh pool-build iso_library
# virsh pool-start iso_library
```

In the preceding example, we used the name **iso_library** to demonstrate how to create a storage pool that will hold ISO images, but you are free to use any name you wish.

4. Verify that the pool (ISO image library) was created:

```
# virsh pool-info iso_library
Name: iso_library
UUID: 959309c8-846d-41dd-80db-
7a6e204f320e
State: running
Persistent: yes
Autostart: no
Capacity: 49.09 GiB
Allocation: 8.45 GiB
Available: 40.64 GiB
```

5. Now you can copy or move the **.iso** images to the **/iso_lib** directory.
6. Upon copying the **.iso** files into the **/iso_lib** directory, refresh the pool and then check its contents:

```
# virsh pool-refresh iso_library
```

```

Pool iso_library refreshed
# virsh vol-list iso_library
Name Path
-----
-----
-----
CentOS8-Everything.iso
/iso/CentOS8-Everything.iso
CentOS7-Everything.iso
/iso/CentOS7-Everything.iso
RHEL8.iso /iso/RHEL8.iso
Win8.iso /iso/Win8.iso

```

7. This will list all the ISO images stored in the directory, along with their path. These ISO images can now be used directly with a virtual machine for guest operating system installation, software installation, or upgrades.

Creating an ISO image library is the de facto norm in today's enterprises. It's better to have a centralized place where all your ISO images are, and it makes it easier to implement some kind of synchronization method (for example, **rsync**) if you need to synchronize across different locations.

Deleting a storage pool

Deleting a storage pool is fairly easy. Please note that deleting a storage domain will not remove any file/block devices. It just disconnects the storage from virt-manager. The file/block device has to be removed manually.

We can delete a storage pool via virt-manager or by using the **virsh** command. Let's first check how to do it via virt-manager:

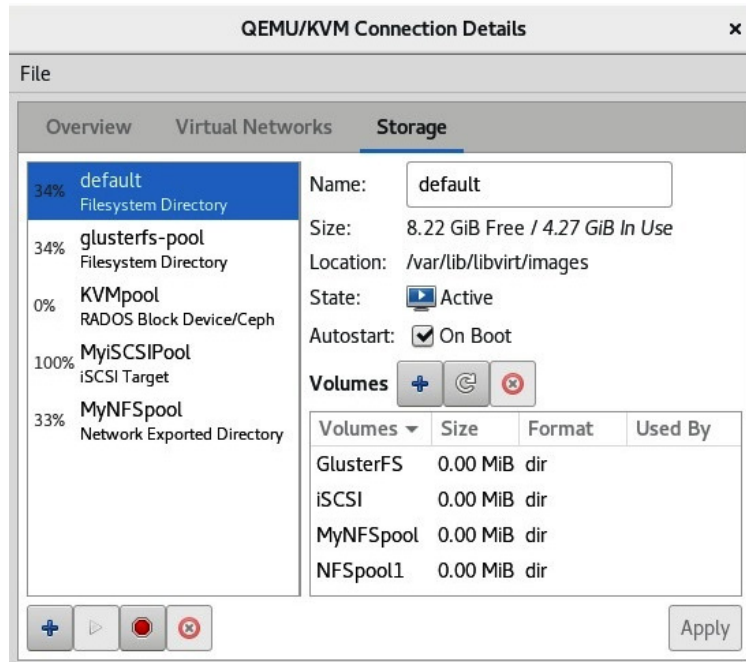


Figure 5.28 – Deleting a pool

First, select the red stop button to stop the pool, and then click on the red circle with an **X** to delete the pool.

If you want to use **virsh**, it's even simpler. Let's say that we want to delete the storage pool called **MyNFSPool** in the previous screenshot. Just type in the following commands:

```
virsh pool-destroy MyNFSPool
virsh pool-undefine MyNFSPool
```

The next logical step after creating a storage pool is to create a storage volume. From a logical standpoint, the storage volume slices a storage pool into smaller parts. Let's learn how to do that now.

Creating storage volumes

Storage volumes are created on top of storage pools and attached as virtual disks to virtual machines. In order to create a storage volume, start the Storage Management console, navigate to virt-manager, then click **Edit | Connection Details | Storage** and select the storage pool where you want to create a new volume. Click on the create new volume button (+):

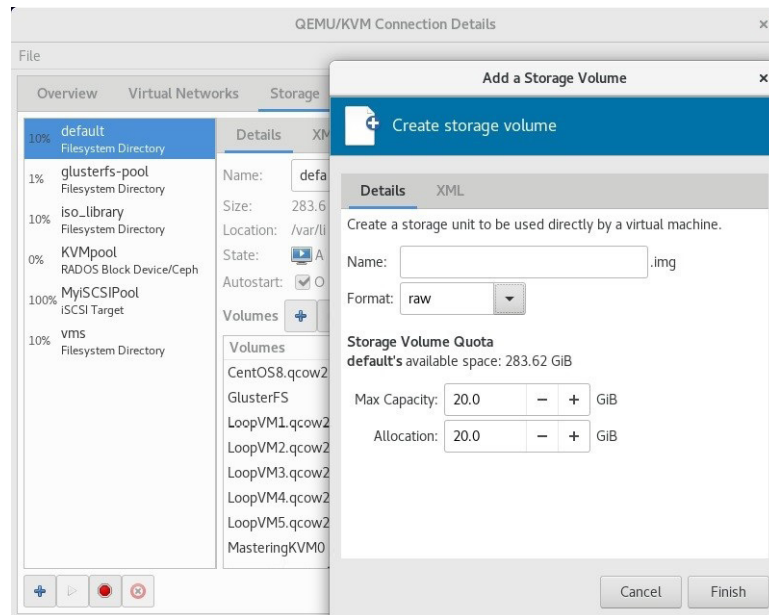


Figure 5.29 – Creating a storage volume for the virtual machine

Next, provide the name of the new volume, choose the disk allocation format for it, and click on the **Finish** button to build the volume and get it ready to attach to a virtual machine. You can attach it using the usual virt-manager or the **virsh** command. There are several disk formats that are supported by libvirt (**raw**, **cow**, **qcow**, **qcow2**, **qed**, and **vmdk**). Use the disk format that suits your environment and set the proper size in the **Max Capacity** and **Allocation** fields to decide whether you wish to go with preallocated disk allocation or thin-provisioned. If you keep

the disk size the same in **Max Capacity** and **Allocation**, it will be preallocated rather than thin-provisioned. Note that the **qcow2** format does not support the thick disk allocation method.

In ***Chapter 8**, **Creating and Modifying VM Disks, Templates, and Snapshots***, all the disk formats are explained in detail. For now, just understand that **qcow2** is a specially designed disk format for KVM virtualization. It supports the advanced features needed for creating internal snapshots.

Creating volumes using the **virsh** command

The syntax to create a volume using the **virsh** command is as follows:

```
# virsh vol-create-as  
dedicated_storage vm_vol1 10G
```

Here, **dedicated_storage** is the storage pool, **vm_vol1** is the volume name, and 10 GB is the size:

```
# virsh vol-info --pool  
dedicated_storage vm_vol1  
Name: vm_vol1  
Type: file  
Capacity: 1.00 GiB  
Allocation: 1.00 GiB
```

The **virsh** command and arguments to create a storage volume are almost the same regardless of the type of storage pool it is created on. Just enter the appropriate input for a **--pool** switch.

Now, let's see how to delete a volume using the **virsh** command.

Deleting a volume using the **virsh** command

The syntax to delete a volume using the **virsh** command is as follows:

```
# virsh vol-delete dedicated_storage  
vm_vol2
```

Executing this command will remove the **vm_vol2** volume from the **dedicated_storage** storage pool.

The next step in our storage journey is about looking a bit into the future as all of the concepts that we mentioned in this chapter have been well known for years, some even for decades. The world of storage is changing and moving into new and interesting directions, so let's discuss that for a bit next.

The latest developments in storage – NVMe and NVMeOF

In the past 20 or so years, by far the biggest disruption in the storage world in terms of technology has been the introduction of **Solid State Drives (SSDs)**. Now, we know that a lot of people have gotten quite used to having them in their computers—laptops, workstations, whichever type of device we use. But again, we're dis-

cussing storage for virtualization, and enterprise storage concepts overall, and that means that our regular SATA SSDs aren't going to make the cut. Although a lot of people use them in mid-range storage devices and/or handmade storage devices that host ZFS pools (for cache), some of these concepts have a life of their own in the latest generations of storage devices. These devices are fundamentally changing the way technology is working and redoing parts of modern IT history in terms of which protocols are used, how fast they are, how much lower latencies they have, and how they approach storage tiering—tiering being a concept that differentiates different storage devices or their storage pools based on a capability, usually speed.

Let's briefly explain what we're discussing here by using an example of where the storage world is heading. Along with that, the storage world is taking the virtualization, cloud, and HPC world along for the ride, so these concepts are not outlandish. They already exist, in readily available storage devices that you can buy today.

The introduction of SSDs brought a significant change in the way we access our storage devices. It's all about performance and latency, and older concepts such as **Advanced Host Controller Interface (AHCI)**, which we're still actively using with many SSDs on the market today, are just not good enough to handle the performance that SSDs have. AHCI is a standard way in which a regular hard disk (mechanical disk or regular spindle) talks via software to SATA devices.

However, the key part of that is *hard disk*, which means cylinders, heads sectors—things that SSDs just don't have, as they don't spin around and don't need that kind of paradigm. That meant that another standard had to be created so that we can use SSDs in a more native fashion. That's what **Non-Volatile Memory Express (NVMe)** is all about—bridging the gap between what SSDs are capable of doing and what they can actually do, without using translations from SATA to AHCI to PCI Express (and so on).

The fast development pace of SSDs and the integration of NVMe made huge advancements in enterprise storage possible. That means that new controllers, new software, and completely new architectures had to be invented to support this paradigm shift. As more and more storage devices integrate NVMe for various purposes—primarily for caching, then for storage capacity as well—it's becoming clear that there are other problems that need to be solved as well. The first of which is the way in which we're going to connect storage devices offering such a tremendous amount of capability to our virtualized, cloud, or HPC environments.

In the past 10 or so years, many people argued that FC is going to disappear from the market, and a lot of companies hedged their bets on different standards—iSCSI, iSCSI over RDMA, NFS over RDMA, and so on. The reasoning behind that seemed solid enough:

- FC is expensive—it requires separate physical switches, separate cabling, and separate controllers, all of which cost a lot of money.
- There's licensing involved—when you buy, for example, a Brocade switch that has 40 FC ports, that doesn't mean that you can use all of them out of the box, as there are licenses to get more ports (8-port, 16-port, and so on).
- FC storage devices are expensive and often require more expensive disks (with FC connectors).
- Configuring FC requires extensive knowledge and/or training, as you can't simply go and configure a stack of FC switches for an enterprise-level company without knowing the concepts, and the CLI from the switch vendor, on top of knowing what that enterprise's needs are.
- The ability of FC as a protocol to speed up its development to reach new speeds has been really bad. In simple terms, during the time it took FC to go from 8 Gbit/s to 32 Gbit/s, Ethernet went from 1 Gbit/s to 25, 40, 50, and 100 Gbit/s bandwidth. There's already talk about 400 Gbit/s Ethernet, and there are the first devices that support that standard as well. That usually makes customers concerned as higher numbers mean better throughput, at least in most people's minds.

But what's happening on the market *now* tells us a completely different story—not just that FC is back, but that it's back with a mission. The enterprise storage companies have embraced that and

started introducing storage devices with *insane* levels of performance (with the aid of NVMe SSDs, as a first phase). That performance needs to be transferred to our virtualized, cloud, and HPC environments, and that requires the best possible protocol, in terms of lowest latency, its design, and the quality and reliability, and FC has all of that.

That leads to the second phase, where NVMe SSDs aren't just being used as cache devices, but as capacity devices as well.

Take note of the fact that, right now, there's a big fight brewing on the storage memory/storage interconnects market. There are multiple different standards trying to compete with Intel's **Quick Path Interconnect (QPI)**, a technology that's been used in Intel CPUs for more than a decade. If this is a subject that's interesting to you, there is a link at the end of this chapter, in the *Further reading* section, where you can find more information. Essentially, QPI is a point-to-point interconnection technology with low latency and high bandwidth that's at the core of today's servers. Specifically, it handles communication between CPUs, CPUs and memory, CPUs and chipsets, and so on. It's a technology that Intel developed after it got rid of the **Front Side Bus (FSB)** and chipset-integrated memory controllers. FSB was a shared bus that was shared between memory and I/O requests. That approach had much higher latency, didn't scale well, and had lower bandwidth and problems with situations in which there's a large amount of I/O happening

on the memory and I/O side. After switching to an architecture where the memory controller was a part of the CPU (therefore, memory directly connects to it), it was essential for Intel to finally move to this kind of concept.

If you're more familiar with AMD CPUs, QPI is to Intel what HyperTransport bus on a CPU with built-in memory controller is to AMD CPUs.

As NVMe SSDs became faster, the PCI Express standard also needed to be updated, which is the reason why the latest version (PCIe 4.0 – the first products started shipping recently) was so eagerly anticipated. But now, the focus has switched to two other problems that need resolving for storage systems to work. Let's describe them briefly:

- Problem number one is simple. For a regular computer user, one or two NVMe SSDs will be enough in 99% of scenarios or more. Realistically, the only real reason why regular computer users need a faster PCIe bus is for a faster graphics cards. But for storage manufacturers, it's completely different. They want to produce enterprise storage devices that will have 20, 30, 50, 100, 500 NVMe SSDs in a single storage system—and they want that now, as SSDs are mature as a technology and are widely available.
- Problem number two is more complex. To add insult to injury, the latest generation of SSDs (for example, based on Intel Optane) can offer even lower latency and higher throughput.

That's only going to get *worse* (even lower latencies, higher throughput) as technology evolves. For today's services—virtualization, cloud, and HPC—it's essential that the storage system is able to handle any load that we can throw at it. These technologies are a real game-changer in terms of how much faster storage devices can become, only if interconnects can handle it (QPI, FC, and many more). Two of these concepts derived from Intel Optane—**Storage Class Memory (SCM)** and **Persistent Memory (PM)** are the latest technologies that storage companies and customers want adopted into their storage systems, and fast.

- The third problem is how to transfer all of that bandwidth and I/O capability to the servers and infrastructures using them. This is why the concept of **NVMe over Fabrics (NVMe-OF)** was created, to try to work on the storage infrastructure stack to make NVMe much more efficient and faster for its consumers.

If you take a look at these advancements from a conceptual point of view, it was clear for decades that RAM-like memory is the fastest, lowest latency technology that we've had for the past couple of decades. It's also logical that we're moving workloads to RAM, as much as possible. Think of in-memory databases (such as Microsoft SQL, SAP Hana, and Oracle). They've been around the block for years.

These technologies fundamentally change the way we think about storage. Basically, no longer are we discussing storage tiering based on tech-

nology (SSD versus SAS versus SATA), or outright speed, as the speed is unquestionable. The latest storage technologies discuss storage tiering in terms of *latency*. The reason is very simple—let's say that you're a storage company and that you build a storage system that uses 50 SCM SSDs for capacity. For cache, the only reasonable technology would be RAM, hundreds of gigabytes of it. The only way you'd be able to work with storage tiering on a device like that is by basically *emulating* it in software, by creating additional technologies that will produce tiering-like services based on queueing, handling priority in cache (RAM), and similar concepts. Why? Because if you're using the same SCM SSDs for capacity, and they offer the same speed and I/O, you just don't have a way of tiering based on technology or capability.

Let's further describe this by using an available storage system to explain. The best device to make our point is Dell/EMC's PowerMax series of storage devices. If you load them with NVMe and SCM SSDs, the biggest model (8000) can scale to 15 million IOPS(!), 350 GB/s throughput at lower than 100 microseconds latency and up to 4 PB capacity. Think about those numbers for a second. Then add another number—on the frontend, it can have up to 256 FC/FICON/iSCSI ports. Just recently, Dell/EMC released new 32 Gbit/s FC modules for it. The smaller PowerMax model (2000) can do 7.5 million IOPS, sub-100 microsecond latency, and scale to 1 PB. It can also do all of the *usual EMC stuff*—replication, compression, dedu-

plication, snapshots, NAS features, and so on. So, this is not just marketing talk; these devices are already out there, being used by enterprise customers:



Figure 3.30 – PowerMax 2000 – it seems small, but it packs a lot of punch

These are very important concepts for the future, as more and more manufacturers produce similar devices (and they are on the way). We fully expect the KVM-based world to embrace these concepts in large-scale environments, especially for infrastructures with OpenStack and OpenShift.

Summary

In this chapter, we introduced and configured various Open Source storage concepts for libvirt. We also discussed industry-standard approaches, such as iSCSI and NFS, as they are often used in infrastructures that are not based on KVM. For example, VMware vSphere-based environments can use FC, iSCSI, and NFS, while Microsoft-based environments can only use FC and iSCSI, from the list of subjects we covered in this chapter.

The next chapter will cover subjects related to virtual display devices and protocols. We'll provide an in-depth introduction to VNC and SPICE

protocols. We will also provide a description of other protocols that are used for virtual machine connection. All that will help us to understand the complete stack of fundamentals that we need to work with our virtual machines, which we covered in the past three chapters.

Questions

1. What is a storage pool?
2. How does NFS storage work with libvirt?
3. How does iSCSI work with libvirt?
4. How do we achieve redundancy on storage connections?
5. What can we use for virtual machine storage except NFS and iSCSI?
6. Which storage backend can we use for object-based storage with libvirt?
7. How can we create a virtual disk image to use with a KVM virtual machine?
8. How does using NVMe SSDs and SCM devices change the way that we create storage tiers?
9. What are the fundamental problems of delivering tier-0 storage services for virtualization, cloud, and HPC environments?

Further reading

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

- What's new with RHEL8 file systems and storage: <https://www.redhat.com/en/blog/whats->

[new-rhel-8-file-systems-and-storage](#)

- oVirt storage:
https://www.ovirt.org/documentation/administration_guide/#chap-Storage
- RHEL 7 storage administration guide:
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/7/html/storage_administration_guide/index
- RHEL 8 managing storage devices:
https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/8/html/managing_storage_devices/index
- OpenFabrics CCIX, Gen-Z, OpenCAPI (overview and comparison):
https://www.openfabrics.org/images/eventpresos/2017presentations/Z_BBenton.pdf