ECE566 Enterprise Storage Architecture Lab #2: NAS, SAN, and Filesystems

Now that we understand RAID, let's use those drives as actual storage.

Directions:

- This assignment will be completed in your groups. However, **every member of the group must be fully aware of every part of the assignment**. Further, while you can discuss concepts with other groups, actual steps and answers should not be shared between groups.
- The assignment will ask for short answers or screenshots; this material should be collected in a PDF file submitted via GradeScope. *Word documents will not be accepted*. Anything you need to include in this document is highlighted in cyan.

1 Prepare your resources

For this assignment, you'll need your server to be configured with a 3-disk software RAID5 with a hot spare, just like in Lab 1 section 3.1. Do this now, and show mdadm command used.

We will again need to set up a filesystem. However, later in this assignment, we'll want to export our RAID device over SAN to a Windows machine, so unlike last time, so we'll need to do a few more steps. Before, we laid our filesystem down directly on the entire block device. This is fine in UNIX, but the more common practice (and one required by Windows) is to have a *partition table*. There are two variants of partition table, the classic MBR-style and the more modern GPT type. We'll use an MBR type for simplicity.

A simple tool to manage MBR partition tables is **`fdisk`**. Research a bit on using this tool (or, if you prefer, one of its more modern siblings, like **`cfdisk`**), and use it to create a single partition on **`/dev/md0`** spanning the entire device. Set the type of this partition to 0x07 (NTFS type). Paste screenshots or console logs of how you did this.

With the partition created, you should now see a new device, **`/dev/md0p1`**, which represents the partition. It's time to put a file system on it, and because we're going to have Windows access this block device, let's pick a filesystem that our Windows client will be able to natively understand: NTFS. Use **`mkfs.ntfs`** to prepare a filesystem on **`/dev/md0p1`**, then mount this new filesystem at the directory **`/x`** (creating the directory if needed). Paste screenshots or console logs of this process.

Place a few files into this filesystem which you'll recognize later. Paste screenshots or console logs showing the files created for later reference.

Later on, you'll also need a **Windows client** (Windows 10 or 11) for which you have administrator access. If your personal computer is Windows-based, you can use it. If you run Mac/Linux on your personal computer, you can either set up a Windows VM in a free hypervisor, such as VirtualBox, or reserve a plain Windows 10 in the Duke Virtual Computing Manager (VCM). The Windows client will need network access to the storage controller.

2 Deploy a CIFS NAS

On the server, install the samba server package, which serves CIFS shares compatible with Microsoft Windows clients. Configure it to share our filesystem in `/x` as a share called "`x`". There are a lot of tutorials on this, so I leave the details to you.

To test it, login to your **Windows client**. In Windows Explorer, navigate to "`\\<IP_ADDRESS_OF_STORAGE_CONTROLLER>\x`". The files you placed there previously should be visible. Verify this, then in Windows, add a few more files.

Provide screenshots or a terminal log of these steps.

Show evidence that you can access the files created in step 1 and note what new files you create.

IN ADDITION, provide the relevant portion of your `smb.conf` file and a screenshot of Windows Explorer showing your storage controller NAS share.

3 Create an iSCSI SAN

Now, instead of a CIFS NAS, we're going to instead deploy an iSCSI SAN. However, we cannot serve the filesystem **`/x`** and the block device **`/dev/md0`** at the same time, since **`/x`** resides on **`/dev/md0p1`**. If we did, both the Linux storage controller and the Windows SAN client would be making changes to **`/dev/md0`**, with neither aware of the other. This would lead to file system corruption and inconsistency.

Therefore, begin by stopping the samba service and using `umount` to unmount **`/x`**. At this point, neither **`/dev/md0`** nor **`/dev/md0p1`** should appear in your mount list:

```
root@xub1404dt:/ # mount
/dev/sda1 on / type ext4 (rw,errors=remount-ro)
proc on /proc type proc (rw,noexec,nosuid,nodev)
sysfs on /sys type sysfs (rw,noexec,nosuid,nodev)
 ...
systemd on /sys/fs/cgroup/systemd type cgroup
(rw,noexec,nosuid,nodev,none,name=systemd)
 (no /dev/md0 in this list)
```

The procedure below will get the iSCSI target software installed and configured.

3.1 Install ‘tgt’

Do the following procedure as root.

First, let’s update/upgrade our existing packages:

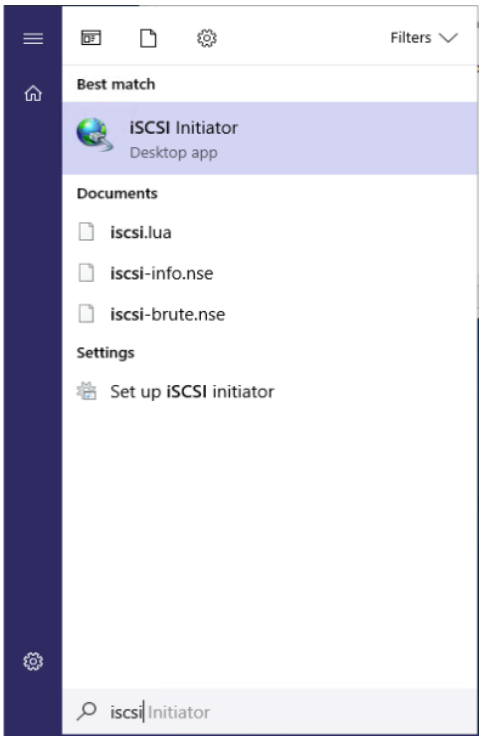
```
# apt update
# apt upgrade
```

Next, install the **`tgt`** package (an iSCSI target)

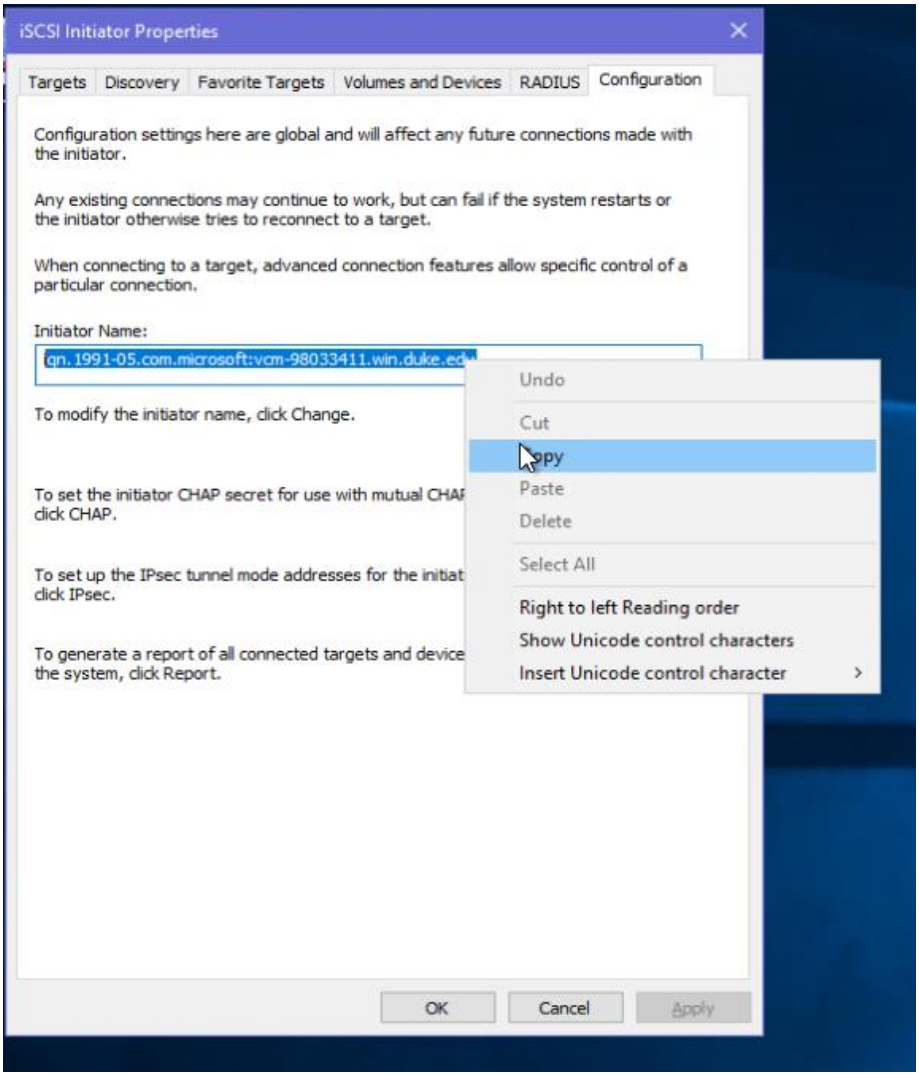
```
# apt -y install tgt
```

3.2 Note the initiator IQN

Via RDP, connect to your Windows VM. In the start menu, search for “iscsi” and launch the “iSCSI Initiator”:



In the “Configuration” tab, make note of the initiator IQN. We could change it to something else if we wanted to, but the default is fine. From now on, we’ll refer to this value as the *initiator IQN*.



Note the initiator IQN in your write-up.

3.3 Prepare target IQN

Let's make up a target IQN for the storage server. Recall the format of an IQN:

```
iqn.(year)-(month).(reversed-domain-name):(arbitrary-string)
```

Where the year and month refer to when the domain was registered. Create an IQN for Duke, noting that duke.edu was registered in June 1986. For the arbitrary string, use your NetID. Note your target IQN in your write-up.

3.4 Set up a LUN export

The **tgt** package keeps its LUN export records in **/etc/tgt/conf.d/**. Create a file called "**mylun.conf**" in this directory. For the content, you can find full documentation on the format here, but the following should be sufficient:

```
<target <TARGET-IQN>>
    backing-store /dev/md0
    initiator-name <INITIATOR-IQN>
</target>
```

Restart the **tgt** daemon:

```
# systemctl restart tgt
```

Display a list of targets to confirm the export worked; the output should look similar to this example:

```
# tgtadm --mode target --op show
```

Include this output in your write-up.

3.5 Attach the LUN

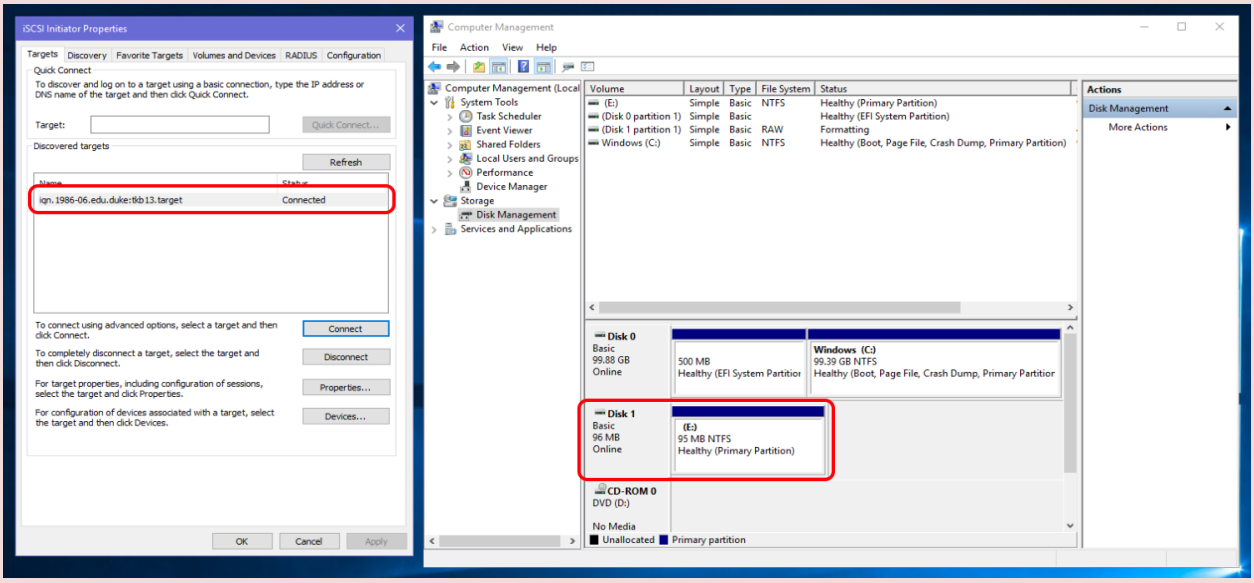
On the Windows VM's iSCSI Initiator controls, in the "Targets" tab, put in the IP address of your storage server into the "Quick connect" target box and hit "Quick connect…". It should identify the target IQN and connect.

You should now be able to open Explorer, go to "This PC", and find a new drive attached. When you view the new drive, you should see the content you put there previously. So we have access to the same data as with the NAS, but now Windows is the one doing the file system logic, so the data shows up as its own drive, and Windows is sending simple read-block/write-block requests to the SAN target.

TROUBLESHOOTING AT THIS STEP

If you get a prompt to format the drive, you may have made a mistake in one of the steps above, as Windows doesn't recognize either the partition table or filesystem.

If no drive appears and you get no prompt, you may wish to check if the block device is attached at all. For this, type "computer management" into the start menu and launch it, then use the "Disk Management" component of this tool. Below is a screenshot showing an attached target in the iSCSI Initiator view next to the block device itself showing in the Disk Management view:



If the target isn't shown on the left, you've failed to connect to the storage server. If you are connected but the block device doesn't appear on the right, the LUN isn't properly attached (export config issue?). If the device exists but says something like "Unallocated" instead of "Healthy", you have a problem with your partition table and/or file system.

Provide screenshots of the Windows GUI steps performed, and evidence that you can access files created both on the Linux server in step 1 as well as files created from Windows via CIFS in step 2.

4 Thin provisioning

NOTE: Each group member should do the "File sizing and sparse files" questions from the individual homework before proceeding.

Thin provisioning is a technique to take advantage of underutilization of storage objects (NAS shares or SAN LUNs). It allows the storage administrator to give out more *logical* storage than the available *physical* storage, banking on the notion that most volumes will be underutilized and that more physical storage can be added as actual utilization increases over time.

We're going to use some basic UNIX concepts to deploy thinly-provisioned iSCSI SAN LUNs.

4.1 Tear down existing SAN

On the Windows client, disconnect from the iSCSI target.

On the Linux server, stop the **`tgt`** service ("**`systemctl stop tgt`**"). Remove the existing LUN export file (you should just move it out of the **`/etc/tgt/conf.d/`** directory so you still have it around as a template for step 4.4).

4.2 Make new container file system

Reformat your RAID device as an ext4 volume. (We could technically reuse the existing NTFS filesystem, but Linux generally prefers its native filesystems, and it's better to start with a clean slate anyway.)

Mount the filesystem to /x like you did in section 2.

4.3 Create virtual LUN backing files

Within this filesystem, make three 1TB sparse files called `lun0`, `lun1`, and `lun2`. We will expose these files as virtual LUNs – files used to represent a logical block device are often called *backing files*.

Check the apparent and allocated size for these files.

4.4 Export and attach the virtual LUNs

Create a new LUN export file based on the earlier LUN export file, except there should be three "backing-store" lines set to each of the files we created. Include your new LUN definition file in your write-up.

Re-start the **`tgt`** service, and on the Windows machine, re-attach to the target. The OS should discover the three LUNs.

4.5 Prepare the LUNs

Unlike in section 3, we didn't prepare a filesystem ahead of time, so the drives won't appear in Explorer. Instead, find them in the disk management interface (described in the troubleshooting of section 3.5). Format each one with an NTFS filesystem. As you do so, be sure to choose "quick format", else Windows will try to write zeroes to the whole thing, then it will no longer be a sparse file. Also, set the filesystem labels to LUN0, LUN1, and LUN2 to help you keep track of which is which. When done, include a screenshot of the Disk management interface showing the three LUNs attached, formatted, and ready.

4.6 Understanding allocation and oversubscription

On the Linux server, check apparent and allocated size for these files.

The allocated size will have increased -- why?

How big are the LUNs as far as Windows can tell?

How much **total** *logical* capacity is the storage server providing to the Windows machine via iSCSI?

How much **total** *physical* capacity does the storage server RAID filesystem have?

What is the rate of oversubscription (logical_bytes_advertised / physical_bytes_available)?

At this point, make note of:

- The free space of the LUN in Windows.
- The allocated and apparent sizes of the LUN file on the server.
- The free space of the underlying storage on the server.

4.7 Add stuff and check the result

On the Windows machine, copy a few megabytes of data into one of the LUNs. After the copy, on the Linux server, monitor the allocated size of the backing file[1]. It may increase for some time after the actual GUI copy appears to finish -- this is due to OS buffering and write-back caching.

When the copy is done and the size change stabilizes, compare to the measurements you made before:

- The free space of the LUN in Windows.
- The allocated and apparent sizes of the LUN file on the server.
- The free space of the underlying storage on the server.

How have these numbers changed?

4.8 Delete stuff and check the result

Delete the data you put on the LUN (hold shift when you do so to force an actual deletion instead of going to the recycle bin; if it does go to the recycle bin, empty the recycle bin before proceeding). Again, check these measurements:

- The free space of the LUN in Windows.
- The allocated and apparent sizes of the LUN file on the server.
- The free space of the underlying storage on the server.

How have these numbers changed?

You should find that Windows says free space went up, but on the Linux server, apparent file size and underlying storage free space are unchanged. This is because the iSCSI initiator has no way to indicate "it's safe to de-allocate these blocks", as block devices don't have a concept of free versus filled. (One exception to this is the TRIM command commonly used on SSDs to inform the drive of de-allocated space, but that is not enabled on a SAN LUN.)

4.9 Add too much stuff and break it

You've probably realized by now that it is not possible to actually fill all three LUNs to their advertised size, as the server doesn't actually have that much capacity. Let's find out what happens when we try. On the Windows host, add stuff to LUN 0, periodically checking the file system free space on the Linux server's RAID and the LUN's allocated size on the server.

You'll eventually fill the physical storage while the LUN appears to have plenty of space. What happens in Windows? Would you characterize the OS as "healthy" right now?

---

[1] Note: you can use the `watch` command to view the output of a command repeatedly.

Check /var/log/syslog for messages. Hopefully the iSCSI target daemon is reporting errors -- note them in your write-up.

Why is running out of space in this manner different than running out of space on a traditional disk? (Hint: could the Windows OS have predicted this would happen?)

As a storage administrator, you should avoid this outcome; what pro-active steps could you take to avoid this while still getting the benefits of thin provisioning?

4.10 Tear it all down

If the Windows machine is just a scratch VM, just delete it. Otherwise, shutdown Windows, stop the **`tgt`** daemon on the Linux server (so Windows doesn't see it on reboot), boot Windows, and detach from the iSCSI target.

On the Linux server, ensure the **`tgt`** daemon is stopped, stop the RAID device, and remove the LUN definition file from **`/etc/tgt/conf.d`**.

5 SAN with NVMe-OF

Let's now try out the relatively new SAN protocol, NVMe-OF. As discussed in class, there are many kinds of networks and hardware that NVMe-OF can work over, but we're going to use our plain commodity Ethernet with TCP. Also, while Linux has robust NVMe-OF support, Windows does not, so for the client device this time, we'll use a VCM instance running Ubuntu Linux.

We'll start on your group server, which will serve as the NVMe-OF target. SSH into your group server. For all of these steps, virtually everything requires root privilege. Rather than type "`sudo`" before every command, just become root by typing "`sudo -i`".

5.1 Learning about /sys

Before we dig into NVMe-OF, let's first observe another mountpoint provided on Linux: `/sys`. As root on your group server, navigate to `/sys` and look around. Then check out `/sys/block`:

```
cd /sys/block
ls -l
```

Hey, it's all the block devices, but they're listed as symlinks! Let's check one out:

```
cd sda
ls
cat size
```

That's the size of `/dev/sda`! In fact, if you were to use `strace` to look at how `lsblk` works, you'll see that it uses these very directories and files to see what block devices are present! So we now know that `/sys` is a place the kernel uses to report info about the system and hardware.

The `/sys` tree is also used to configure kernel features. Let's check that:

```
cd /sys/kernel/config
ls
```

Okay, there's a few things here, but nothing we recognize or need to mess with. This is because support for acting as an NVMe target is not yet loaded into the kernel. For that you'll need to learn about…

5.2 Kernel modules

Most modern kernels are modular, including Linux. This means that software components can be loaded and unloaded into the OS kernel as needed. You can list them with:

```
lsmod
```

Run this and check out the modules loaded.

As for NVMe, Linux already comes with a kernel module for acting as an NVMe target. We can load it with the `modprobe` command. Let's load the core NVMe target module, as well as a support module for using TCP as our transport protocol. (A module for RDMA is also provided, but we won't be using it.)

```
modprobe nvmet
modprobe nvmet-tcp
```

You can confirm these modules are loaded by running "`lsmod | grep nvme`". Note the output in your write-up.

5.3 Reading the kernel log

Later on, it will be important to read the kernel log to see what stuff is happening in the background. For reference, you can do so with this command:

```
dmesg
```

Run it now and just take a look at the kind of stuff it's telling you.

5.4 Setting up the NVMe target: the subsystem

Now, head let's check out `/sys/kernel/config` again now that we've loaded those modules:

```
cd /sys/kernel/config
ls
```

An `nvmet` (NVMe Target) directory has appeared! Let's check it out:

```
cd nvmet
ls
```

Hey, it's the three NVMe-OF concepts from class: `hosts`, `ports`, and `subsystems`! As a reminder:

- "Host" means *client*, also known as *initiator*.
- "Subsystem" means *server*, also known as *target*. This is what we're setting up now on your group server.
- "Port" is a method of connecting; this will be an IP address and TCP connection for us.
- "Namespace" is the term for an actual block device exported, equivalent to "LUN" in iSCSI/FCP.

Let's set up this subsystem. First, we need to pick an NQN (NVMe Qualified Name) for the server. Similar to the IQN, we'll use the duke.edu domain, the year/month it was registered, your NetID, and an arbitrary string. We combine them to form "`nqn.1982-06.edu.duke.`*`YOURNETID`*`.subsys1`". Note your NQN in the write-up. Once nice thing about this driver is that it uses the virtual `/sys` filesystem for all its configuration, so we're going to create a subsystem by literally just making a directory:

```
cd subsystems
mkdir nqn.1982-06.edu.duke.YOURNETID.subsys1
cd nqn.1982-06.edu.duke.YOURNETID.subsys1
ls
```

As you can see, several "files" appeared to allow configuration of this subsystem!

One thing we'll do to make this lab shorter is to turn off access permissions and allow all clients to connect. This generally shouldn't be done on a public network, but we won't store anything of value on our SAN, and it won't be around very long. (We could configure allowed hosts, but this requires setting up a cryptographic authentication protocol called DHCHAP, which we're skipping.) We allow all access with:

```
echo 1 > attr_allow_any_host
```

Next, let's set up the namespace (block device) that we want to export. From your subsystem directory:

```
cd namespaces
mkdir 1
cd 1
ls
```

Here we can see configuration files for the namespace. We only need to set which device will map to this namespace. We could use an image or a software RAID like before, but let's make it simple and just use the entirety of our first hard disk drive. For me, this was `/dev/sdb`, but you should check lsblk to figure out yours! **Do not export your SSD, or your OS will be destroyed and you'll have to reinstall from scratch!** Once you know your HDD device, configure the namespace to use it as follows:

```
echo /dev/sdb > device_path
echo 1 > enable
```

5.5 Setting up the NVMe target: the port

Because of the variety of protocols and networks that NVMe-OF can work over, we have to configure the port separately.

```
cd /sys/kernel/config/nvmet/ports
ls
```

You should find nothing there yet. We'll make a port using `mkdir`, similar to the namespace:

```
mkdir 1
cd 1
ls
```

Let's configure this port to listen on IPv4 address 0.0.0.0 (meaning it listens to all IPv4 addresses it has), via TCP on port 4420. To do so:

```
echo 0.0.0.0 > addr_traddr
echo tcp > addr_trtype
echo 4420 > addr_trsvcid
echo ipv4 > addr_adrfam
```

Lastly, we have to indicate that the subsystem we just configured will be using this port. To do this, we actually use symlinks[2]!

```
cd subsystems/
ln -s /sys/kernel/config/nvmet/subsystems/nqn.1982-06.edu.duke.YOURNETID.subsys1 .
ls -l
```

At this point the NVMe-OF target should be ready. Run `dmesg` and you should see messages like this:

```
[ 1320.846639] nvmet: adding nsid 1 to subsystem nqn.1982-06.edu.duke.tkb13.subsys1
[ 1406.210296] nvmet_tcp: enabling port 1 (0.0.0.0:4420)
```

Screenshot your version of the above and include it in your write-up.

5.6 Setting up the NVMe host

For the client ("host"), we'll use an Ubuntu Linux VM from the Duke VCM facility. **Reserve one if you don't have one already, and SSH into it now**. *Do not do these commands on your group server – we're setting up the other side!* As before, use "`sudo -i`" to become root. Let's fully update the machine, then install the NVMe client command line tools:

```
apt update
```

[2] It is totally perfect for this course that the driver works like this. We *just* learned about symlinks in class, and this driver happens to use them as a core part of its configuration. I didn't plan this – it's just an elegant coincidence!

```
apt dist-upgrade
apt install nvme-cli
```

Next, we need to install the kernel module support for acting as an NVMe-OF host. Note the name change from before – this is "`nvme-tcp`", not "`nvmet-tcp`"! Load it and use lsmod to confirm:

```
modprobe nvme-tcp
lsmod | grep nvme
```

Show the successful loading of the module in your write-up.

At this point, make note of the IP address of your group server – you can find this by typing "`ip addr`" on the target, or looking it up in the server inventory sheet.

On the host (client), let's scan the target for namespaces:

```
nvme discover -t tcp -a TARGET_IP_ADDRESS -s 4420
```

You will probably see two log entries, a generic discovery namespace with an NQN like "`nqn.2014-08.org.nvmexpress.discovery`", and the real namespace we created, "`nqn.1982-06.edu.duke.`*`YOURNETID`*`.subsys1`". Show your discovery output in your write-up.

Okay, let's connect to it and get our namespace!

```
nvme connect -t tcp -n nqn.1982-06.edu.duke.YOURNETID.subsys1
  -a TARGET_IP_ADDRESS -s 4420
```

If successful, there will be no output, but we can see our block device with "`lsblk`". You should now see something like "`nvme0n1`" listed. We know enough about NVMe now to guess at the meaning of this name – it's NVMe subsystem 0, namespace 1, hence "`nvme0n1`". You can also confirm that its size matches that of the disk in your server. Show your block device listed in lsblk writeup.

5.7 Using the block device to set up an experiment

Note that, at any time, you can disconnect with this block device with:

```
nvme disconnect -n nqn.1982-06.edu.duke.YOURNETID.subsys1
```

Don't do so yet, but note that this ability gives is a great chance to test the effect of **filesystem journaling** in practice, since this "disconnect" is equivalent to ripping the drive out during a write test.

We're going to set up an experiment where we interrupt writes to a journaled and non-journaled filesystem, then observe the effects this has on recovering the two filesystems.

In order to have two filesystems on this block device at once, we'll use partitioning. This time, we'll partition with `cfdisk` using a GPT partition type:

```
cfdisk /dev/nvme0n1
```

In the interface, create two partitions, each 30GB. Then choose “write”, confirm by typing “yes”, and quit.

Now, when you run `lsblk`, you should see output like the following, indicating the detection of two partitions of our NVMe-OF block device:

```
nvme0n1                 259:1    0    72G  0 disk
|-nvme0n1p1             259:2    0    30G  0 part
`-nvme0n1p2             259:3    0    30G  0 part
```

Show this in your write-up.

Now we will make a filesystem on each of these partitions. For the first partition, we’ll use ext2 (which we learned about in class), and on the second, we’ll use ext3 (which is basically ext2 plus journaling). Create the filesystems:

```
mkfs.ext2 /dev/nvme0n1p1
mkfs.ext3 /dev/nvme0n1p2
```

Further, it will be interesting to put the ext3 filesystem into data journaling mode, which will reduce performance but improve data retention on crash:

```
tune2fs -o journal_data /dev/nvme0n1p2
```

Now, let’s set up two mountpoint directories. Traditionally, arbitrary mounts live in `/mnt`, so we’ll go there and make two directories called “1” and “2” under there:

```
cd /mnt
mkdir 1 2
ls
```

Now we can mount our two filesystems:

```
mount /dev/nvme0n1p1 1
mount /dev/nvme0n1p2 2
```

At this point, you should play around in each filesystem, confirming you can read and write files within each. As a fun test, make a file with an interesting, unique name in each of the two filesystems. Then, on the target (your group server), use `grep` to confirm that these unique strings appear within the block device being served via NVMe-OF (e.g., /dev/sdb). Show this process and its results in your write-up.

5.8 The write test

First, before we start our test, run `dmesg` and note the state of the kernel log. Later on, we’ll look at the log again to see the write errors that are going to appear.

We want to generate write output to each filesystem with content that is predictable. We can use the standard command `seq`, which print integers, to do so. First, run:

```
seq 15
```

As you can see, it prints that many integers.

We're going to do the following:

- Run two instances of "`seq`" to write to a file in each of the two filesystems.
- Wait about 30 seconds.
- Disconnect the NVMe-OF block device while it's running.

Since this test involves a modest amount of timing, be sure to read and understand the steps that follow before you start so you can do them reliably.

Start up the write processes. The '`&`' at the end means "background", so the command will be happening as you proceed with the rest[3].

```
seq 10000000000 > /mnt/1/testfile &
seq 10000000000 > /mnt/2/testfile &
```

Wait about 30 seconds. Now, disconnect the subsystem:

```
nvme disconnect -n nqn.1982-06.edu.duke.YOURNETID.subsys1
```

At this point, your two filesystems will have a very bad time as their block device disappears! View the kernel log, and note all IO errors and related events in your write-up:

```
dmesg
```

At this time, you can reconnect the subsystem so we can assess the damage:

```
nvme connect -t tcp -n nqn.1982-06.edu.duke.YOURNETID.subsys1
  -a TARGET_IP_ADDRESS -s 4420
```

Use `lsblk` to see your block device re-appear. Show the output in your write-up. Note: it may have been assigned a new namespace number, so what once was "nvme0n1" may now be "nvme0n2" or similar – this is okay.

Don't mount the filesystems yet! We need to use `fsck` (Filesystem Check) to identify and correct inconsistencies that resulted from the rude disconnection of the block device. Normally, this tool operates interactively, asking you about each fix it wants to do. We'll apply the "`-y`" option to auto-confirm. This way, we can prepend the "`time`" command to see how long each recovery takes.

```
time fsck -y /dev/nvme0n2p1
time fsck -y /dev/nvme0n2p2
```

[3] If you aren't familiar with background processes, take a few minutes to learn before you proceed. You should understand the "`jobs`" command, and the use of "`kill`" to end jobs early. Also, while you don't need it for this lab, it's wise in general to know about the Ctrl+Z shortcut and the "`fg`" and "`bg`" commands.

Include the full output of these commands in your write-up.

Once these are complete, remount the two filesystems, then **answer each of the questions below in your write-up**:

1. How long did each recovery take in seconds?

2. Assuming that the first device was much slower than the second, why is that?
   (If the first device wasn't slower, contact the instructor to help investigate – it really should be!)

3. The output of each `fsck` command shows each "fix" it applies. How many fixes were needed for the first device (no journal)? For the second (with journal)?

4. Examine the two test files. Did either of them disappear? If they're both present, use "`tail`" to examine the end of each – did any corrupt data get appended? If both files are present, which file is larger, and by how much? Why do you suppose that is? (Hint: recall that partition 2 did data journaling. If you aren't sure of the answers, talk with the instructor.)

5.9 All done

We need to tear all this NVMe-OF stuff down, but here's a handy shortcut: all the config changes we made were in-memory only. So rather than unwind all that config, you can just reboot the target (group server) and host (VCM VM), and it will all be gone. Do so now:

```
reboot
```

Good job, you finished Lab 2!