PRACTICAL PORTS

Setting up an NFSv4 Fileserver on OpenZFS

BY BENEDICT REUSCHLING

We recently experienced a small disaster at work that got me re-thinking our current file-serving strategy. We deploy distributed applications like MongoDB, Hadoop, Spark and others via Ansible. The binaries to install these don't come from a software repository but are downloaded from the vendors website because of the up-front registration to get the Enterprise edition with extra features. The archives that contain these binaries are quite large (even compressed) and copying them from the Ansible controller to the target machines over the network usually takes some time.

A while ago, we figured there could be a faster way to do this: put all those binaries into a network share (coming from Ceph in this case), mount them on the target machine, and then point Ansible to them. For Ansible, this looks like the files are "local" to the target machine, so it installs right from the share. Even in those instances where we still need to copy the files to a local directory (in case of Hadoop or Spark, which is nothing more than an archive to extract), it's much faster than transferring them from the Ansible controller first.

When running the script again, things went wrong…

One of the last actions in the deployment playbook is to unmount the share. The share gets mounted as read-only so that no accidents can happen, and we simply don't need write access during the deployment. Even if the share does not unmount properly and may still be around when our students start using the server, they cannot delete any important files because of the missing w-bits.

All was fine and good until one fateful day when one of our student helpers working on a new version of the NoSQL databases needed to put a newer version of said software on the share. It was easy enough to mount the share read-write (they are allowed to do that as part of their job) and put the new binaries next to the others. When running the script again, things went wrong: the playbook ran and faithfully did its work. What I get later is an email from the student telling me that the script mounted the share (still read-write) and did some cleanup

action afterwards. However, what it did not check for is whether the share was cleanly unmounted from the system. Which did not happen in this case. So, the cleanup job happily ran over the still mounted Ceph share and thoroughly cleaned all the files in there. Oops!

The email from the student asked whether I still had an older snapshot from the Ceph share to restore the files. I did not, as the share was provided by our IT department. When inquiring there directly, it turned out that they did not backup that share at all. A new backup system was put in place recently but was not ready yet to do the backups. There were also no other backups available, even though the Ceph share was replicated among three separate buildings on campus. No luck there.

I was rather calm about this for several reasons: first, this could have easily happened to me, second, the files could be restored and nothing on the share was irreplaceable. Third, this provided me with an opportunity to make things more robust in case this happened again in the future, which is the best takeaway from events like this.

I updated the playbooks to do an extra check after the unmount task to determine whether the share was actually unmounted. If not, it would do a forced unmount, and if that also failed, then the playbook execution would stop at that point. There are usually good reasons for a network share to not unmount properly (typically when there are some files still accessed), but not continuing would be better than risking another accidental data erase.

Since I did not have full control over the Ceph share and my network share needs are not that big, I decided to run my own. Instead of Ceph, I chose FreeBSD's ZFS with its integrated NFSv4 sharing. That way, I could run regular snapshots which don't grow too much when there are no changes on a mostly read fileshare. Also, instead of relying on simply mounting the share read-only, I could set the ZFS property of the same name to "`on`." With `readonly=on`, not even the root user would be able to remove files on a dataset with that property. Plus, I could get the regular data-integrity checks that ZFS does and maybe some space savings from compressing that dataset.

Implementing the Solution

Here are my notes setting up the NFSv4 server on a FreeBSD system. First, I went to `/etc/rc.conf` and added the following lines:

```
nfs_server_enable="YES"
nfsv4_server_enable="YES"
nfsuserd_enable="YES"
hostid_enable="YES"
rpcbind_enable="YES"
mountd_enable="YES"
rpc_lockd_enable="YES"
rpc_statd_enable="YES"
```

This enables the NFS server in general, ensures proper permissions using `nfsuserd`, and does correct locking for the RPC calls that NFS makes. Next, I checked that `/etc/exports` contained only the following line:

```
V4: /
```

This is telling the NFS server on FreeBSD that it should use NFS version 4, but that the actual definition of what paths are shared will be determined by ZFS. I've read on the web that this file could even be empty now, but it does not hurt to have it in there either.

I created the ZFS dataset that will do the file sharing like any other:

```
zfs create -o atime=off zroot/fileshare
zfs set mountpoint=/fileshare zroot/fileshare
```

The nice thing about ZFS is its inheritance. If I decide to create a sub-dataset below fileshare that should also serve NFS, I don't have to separately configure it, as it already inherits all properties from the parent (except the mountpoint). If I don't want to share the sub-dataset, then I can just as easily disable the sharing by setting the `sharenfs` property to off (which is the default).

Let's first copy some files over to the NFS share and then set the `readonly` property to "on" (that's what we came here to do in the first place):

```
cp /some/important/files /fileshare
zfs set readonly=on zroot/fileshare
```

Clever ZFS users could take this one step further by taking a snapshot of the share, mounting that into the system and then sharing that over the network. This also ensures that the files can't be changed, as ZFS snapshots are read-only in nature. But I'll leave that as an exercise to you for another day.

In the `sharenfs` property, I can define all the parameters that I would normally need to put into a separate NFS config file. That way, the information about how to share this dataset over NFS stays with it even when it gets sent to a different pool. This all-in-one-place nature of ZFS is making configuration much simpler, as there are fewer places to look for errors.

In my case, I only wanted to share the NFS on a certain subnet. You can also list hostnames or IP-addresses separated by commas instead. That way, you limit who can mount the share to a number of hosts for some extra security.

```
zfs set sharenfs="-network 192.168.0.0 -mask 255.255.255.0
-maproot=user,-alldirs" zroot/fileshare
```

The `maproot=user` part defines that if a user accesses the share and there are files with the user's permissions, then the server maps them to the same local permissions, even though they may be different on the server. For example, Joe may have a local `uid/gid` of 2000, while on the NFS server, the users all begin starting at 3000. The NFS server will have Joe's files set with `uid/gid` as 3000, but when Joe accessed the share, he will see his familiar 2000 of the local system to not get confused. The `-alldirs` option allows mounting at any directory within `/fileshare`. Find out more about these and other options by reading exports(5).

That's all for the server part. We need to start all the services listed in `/etc/rc.conf` to start sharing the mounted dataset:

```
service nfsd start
service mountd start
service nfsuserd start
```

Some of these services should be automatically started with the NFS server, but carefully check the status output from each of these services to see that they are running. The output of

```
sockstat -4l
```

as well as

```
rpcinfo
```

and

```
nfsstat
```

help you determine any problems when the share won't mount for some reason.

Another way to see the currently shared datasets is running

```
cat /etc/zfs/exports
```

to see the whole list.

Next, we look at the client. I use FreeBSD and Ubuntu Linux systems to mount the share and describe what each needs to access it. Starting with the FreeBSD client, it needs only a few lines in `/etc/rc.conf`:

```
nfsuserd_enable="yes"
hostid_enable=YES
nfscbd_enable=YES
```

The NFS user daemon takes care of the mapping of user ids and groups from the server as described above. The `hostid` uniquely identifies this system to the NFS server and the NFS callback daemon handles callback requests from the server. The man page for it assures me that mounts will work without it, but it won't hurt to activate it right away so as not to scratch my head about it later.

Starting those services right away, we can look at what the NFS server (called `myfiler`) is offering to us:

```
showmount -e myfiler
```

This should give us a list of exported shares that we can mount. From the command-line, the mount command is invoked as follows:

```
mount -t nfs -o nfsv4 myfiler:/fileshare /media
```

If you like to have this share mounted each time your system starts, add it to `/etc/fstab` like this:

```
myfiler:/fileshare          /media        nfs      rw,tcp,noatime,nfsv4 0 0
```

The `noatime` and `rw` options are not strictly necessary as we took take care of them from the ZFS side earlier, but the `nfsv4` must be there to let the system know that it is talking to version 4 of NFS.

At this point, you should be able to mount the share, see the files in there together with the correct user and group IDs.

Over on a Ubuntu Linux system, we first need to install the NFS server bits as they are not part of the base system:

```
apt install nfs-common
```

Use the package distribution of your particular distribution, the rest of the setup should be the same from here on. It turns out that this is all that's needed. Mounting the share on the command line is done via:

```
mount -t nfs -onfsvers=4 myfiler:/fileshare /media
```

Of course, mounting can happen to any other local directory that exists, not just to /media. I just use it because it exists and is usually empty. Mounting over an existing directory will hide its contents until the next time the NFS share is unmounted again. Ensure that you don't do this on any vital directories that this system needs to run properly. Whatever directory you picked, in case you also want to have the share mounted each time the Linux system boots, put this line into `/etc/fstab`:

```
myfiler:/fileshare          /media        nfs      rw,nfsvers=4     0  0
```

That's all. The server that I put in place will regularly copy the contents from the NFS share to the Ceph to have an extra backup. But I worry less now that ZFS is backing my files: regular snapshots and the `readonly` property should avoid future mistakes like the one described above.

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. In the past, he served on the FreeBSD core team for two terms. He administers a big data cluster at the University of Applied Sciences, Darmstadt, Germany. He's also teaching a course "Unix for Developers" for undergraduates. Benedict is one of the hosts of the weekly bsdnow.tv podcast.