PRACTICAL PORTS

Importing a ZFS ZIL via iSCSI

Don't do this at work – like I did

BY BENEDICT REUSCHLING

This column covers ports and packages for FreeBSD that are useful in some way, peculiar, or otherwise good to know about. Ports extend the base OS functionality and make sure you get something done or, simply, put a smile on your face. Come along for the ride, maybe you'll find something new.

I have run our CS department's PostgreSQL server on FreeBSD in a virtual machine for a number of years now with great success. The server is mainly used in the database classes and for projects requiring a database backend. I gave a talk at vBSDcon 2019 about the server which you can find on youtube.

Recently, the department that hosts the virtualization server for this machine changed their underlying storage to Ceph. This added more capacity and redundancy for them by synchronizing the I/O between three different buildings on campus. Around the same time, the database professors devised new lab exercises to let students become familiar with large sets of data. One of the exercises was to create mass data and insert it into a database table, measuring execution time with and without a table index. All well and good, but soon after that particular lab began, professors and students started complaining about poor performance. In some instances, a local postgres installation on students' laptops ran faster than on our server with more CPU and memory. For example, running a "`SELECT COUNT(*)` from `bigtable;`" with roughly 10 million rows took 2 minutes and five seconds on average. A local laptop took about a second. Running the same query a second time took 1 second on the server, proving that it was served from the much faster main memory cache.

Soon after that particular lab began, professors and students started complaining about poor performance.

I started my investigation on postgres, tuning some parameters in `postgresql.conf` and restarting the server. This had only marginal success and people still complained about long insert and query times. Since there was proof that PostgreSQL's default settings had better performance, the problem must have been storage—or I/O-related. When the VM was created, its underlying portion of the Ceph storage was transformed into a ZFS pool, which in turn provided most of it as a dataset for the postgresql database. Since a lot of students were inserting the same data and queried it afterwards, the ZFS ARC was serving those directly from memory. Not all data could fit in the ARC or was evicted from it by other queries. As soon as we hit the disk with writes, the slowdown was noticeable with large data generated by the users.

To confirm our suspicion that the underlying storage was the problem, I picked a server from our big data cluster with 64 CPUs, 384 GB RAM, 4x 512 GB NVMe and installed FreeBSD on it. Then I used "`zfs send`" to copy the dataset hosting the `postgresql` server over to this new server. After starting the `postgres` service, I had a complete copy of the server to play with on beefier hardware. Running the same `COUNT(*)`-queries on the new server proved that they were as fast (if not faster) than a student's laptop, even if they had an SSD. Clearly, performance on our virtual server was to blame. Solving this problem was not that easy though as our IT-department couldn't simply attach an SSD or NVMe to this VM to speed it up. Purchasing and installing it in the server (which meant downtime) would take longer than the remaining time in the semester.

Solving this problem was not that easy though as our IT-department couldn't simply attach an SSD or NVMe to this VM to speed it up.

My idea was to export one of the NVMe disks from the server we just tested on to the VM via iSCSI to create a tablespace. Tablespaces allow the database administrator to define where database objects should be stored on the file system. With iSCSI, storage from a server (called target) can be sent over the network to another machine (called initiator) that imports it. Instead of a network share, the iSCSI protocol lets the storage appear on the importing machine as local block storage—an important difference. This new storage is handled like any other and can be partitioned and formatted with a new filesystem just like a device attached locally.

FreeBSD has iSCSI built-in by default and only requires a few changes in configuration files to set it up. Here is the configuration on the server exporting the NVMe:

First, I created a volume of 200 GB on one of the NVMe drives called iscsi_export:

```
# zfs create -V 200g nvme/iscsi_export
```

Next, I edited /etc/ctl.conf to contain these sections:

```
portal-group pg0 {
        discovery-auth-group no-authentication
        listen ip.address.of.initiator
}

target iqn.dns-name-of-initiator:nvme {
        portal-group pg0
        chap postgres verysecurepasswordgoeshere

        lun 0 {
                path /dev/nvme/iscsi_export
                size 200G
        }
}
```

I changed the ownership and permissions on this file to `root` since it contains a cleartext password.

Upon reboot of the server, the iSCSI initiator should be started again, so I put `ctld_enable="YES"` into `/etc/rc.conf`:

```
# sysrc ctld_enable=yes
```

To activate the initiator, I started the service:

```
# service ctld start
```

This mostly follows the descriptions of the iSCSI section in the FreeBSD handbook.
Over on the VM importing the storage disk, I put the following into `/etc/iscsi.conf`:

```
        TargetAddress    = ip.address.of.initiator
        TargetName       = iqn.dns-name-of-initiator:nvme
        AuthMethod       = CHAP
        chapIName        = postgres
        chapSecret       = verysecurepasswordgoeshere
}
```

Since the `postgres` users log into this server via SSH to run postgresql's commandline utility `psql`, keeping the password in this file secure from prying eyes is important. A `chmod` of 0700 followed by a `chown` with owner and group set to `root` and `wheel` solves this. An entry to `/etc/rc.conf` is necessary to initiate the storage import upon reboot (more on that later):

```
# sysctl iscsid_enable=yes
```

Next, we can import the disk by starting the service:

```
# service iscsid start
```

Upon successful import, a new device (probably `da0` or similar) appears in `/dev`. A separate ZFS pool was created on it:

```
# zpool create nvme_ts /dev/da0
```

Yes, this is not redundant, but for our benchmarking purposes, it was sufficient enough. On the postgres side, logged in as the database superuser in `psql`, the tablespace is defined by this statement (see https://www.postgresql.org/docs/current/manage-ag-tablespaces.html for details):

```
psql#>CREATE TABLESPACE nvme LOCATION '/nvme';
```

Checking the access permissions again, but after the command is complete, the postgres database users can use the tablespace and put database objects (like tables) on it. Either by explicitly defining where the data should be stored:

```
psql#>CREATE TABLE nvme_powered_table(i int) TABLESPACE nvme_ts;
```

or setting the tablespace as default:

```
psql#>SET default_tablespace = nvme_ts;
```

With this new configuration (clearing the cache first) and reload of a fresh batch of 10 GB data into the `nvme_powered_table`, the database insert performance on the VM improved to 7 seconds (from its original more than 2 minutes). Having an NVMe tablespace is certainly nice, but we went further. This is also when trouble started...

Not Thinking Things Through

We decided to use the exported storage as a ZIL to speed up the slower writes on the Ceph-backed pool. The ZIL would acknowledge to the application (the database) that the writes have reached stable storage and would later write to the slower disk while the database continued its work. A ZIL usually does not have to be big, as the data in it gets quickly evicted. We reduced the amount of exported disk space in the `iSCSI-initiator` and re-imported the disk in the database VM. Then we configured the iSCSI disk as a ZIL with the following command:

```
# zpool add pgpool log da0
```

The device showed up and worked immediately. I/O on the pool was now quickly acknowledged as "written" and the database could continue without waiting. The ZIL trickled the write

requests to the slower Ceph storage. This boosted the database performance a good degree and we went into production.

Don't Try This At Work

What I did not realize at the time is how badly this integrates into the boot process. When FreeBSD with a ZFS-only filesystem boots, it tries to detect all the storage devices contained in the pool. At this point during the boot, the network is not yet completely configured and thus no iSCSI services are available to import the external device. When it comes to the ZIL device, it turned out that ZFS requires this to boot properly and complains about a missing disk in the pool. The boot process is halted at this early stage, even though the main vdev of the pool was available (but ZIL wasn't). You can imagine that this does not go well on a production server and only the management console of the server itself revealed what was going on.

Note that this can happen in two ways: either the iSCSI target (the server exporting the storage) goes down or loses connectivity, or the initiator (the client importing the device). Seasoned sysadmins know that during a typical day, interruptions of this kind can happen, often unannounced and unexpected. It is only a matter of time when this would have happened and now that it did, we needed a way to fix it--quickly.

Rebooting the server with a FreeBSD ISO image and selecting the Live-CD option in the installer was next. From the Live-CD's shell environment, we could mount the pool with the missing ZIL device on `/mnt` like this:

```
# zpool import -R /mnt -m pgpool
```

After the import was finished, we could inspect the remaining devices in the pool:

```
# zpool status
```

The output showed the missing cache device with its long unique numeric identifier. The next action was to remove the ZIL device from the pool:

```
# zpool remove pgpool <verylongnumericidentifier>
```

Typing in the long identifier instead of the much shorter device name serves as a good reminder to avoid this situation in the future. Once this had been done and the output of `zpool status` confirmed the removal, the pool was exported again. This is usually done upon reboot, but we did not want to take any chances.

```
# zpool export pgpool
```

After the machine rebooted, we were happy to see it complete the boot this time and gave us our familiar login prompt back. Disaster averted, but the underlying performance problem was still present.

Happy Ending

Clearly, the iSCSI export is too risky and could fail again. Although we did run like this for a whole semester, Murphy's law will let that happen at the worst time of night when sysadmins are supposed to be sleeping. Certainly, a script could safely remove the ZIL from the pool upon every shutdown. But power losses or crashes on both machines involved in the iSCSI export are not covered by this. Luckily, our IT department was finally able to provide us an SSD-backed Ceph storage as an alternative for this machine. The import is similar to iSCSI but is more stable and less prone to crashes.

Ceph on FreeBSD works, but importing this device proved to be...interesting. Ceph supports this kind of import on FreeBSD only via geom_gate, which is similar to iSCSI. After installing the `net/ceph14` package, the `rbd-ggate` command was available (rbd is the Rados Block Device of Ceph). The man page rbd-ggate(8) is rather short, listing only a few commands and switches. I was a bit worried at first as it dates back to 2014. With no recent updates, chances are that support could have been broken by a change on newer FreeBSD versions. This was unfounded, however. We only had to deal with some of the differences in how Linux and FreeBSD deal with commandline arguments. On Linux, a `--option` is used, whereas on FreeBSD a single `-option` is more common. The command initially looked like this:

```
# rbd map -t ggate volumes/ssdvolume
```

The `volumes/ssdvolume` is the path to the SSD ceph storage given to us by the IT department and maps a geom gate device upon successful import. The command failed because the `--id` of the user doing the import was not provided (username and password protects this storage from unauthorized imports by others). Here's where the mixing of single and double dashes became problematic, as the Linux-based rbd command refused to mix the `--id` with the single `-t` parameter. We found a solution by providing the ID as an environment variable like this:

```
CEPH_ARGS='--id postgresdb' rbd map -t ggate volumes/postgresdb
```

With this combination, the command ran successfully and told us

```
ggate0 created
```

This was confirmed by looking at `/dev/ggate0`. This is the imported device from Ceph, on which we could now create a new ZFS pool:

```
# zpool create ssdpool /dev/ggate0
```

Remembering what we learned from last time, we tried rebooting the machine to see how it coped with this device during boot. We were happy to see that the system did reboot without issues, and we could then re-import this new pool using:

```
# zpool import ssdpool
```

We could then create a little startup script that was executed once the system finished booting to automatically re-import this pool and activate the postgres database on it. The postgres database was cloned by snapshotting and zfs sending from the old, slower pool and receiving it on the faster ssdpool. This works quite well, and the performance difference is definitely noticeable. As I write this, the first student groups are already working on it (without their knowledge) and I have not received any complaints yet.

Lessons Learned

Measure where performance is lost and isolate the bottlenecks. Use different test cases to confirm any hypotheses about where the problem might be located. Test things before putting them into production. Ensure solutions survive a reboot of both the exporting and importing machine when dealing with storage coming over the network. Keep a FreeBSD Live-CD ISO image handy to fix things in case of disaster. Document every step and command for yourself and your peers to have them available when people are breathing down your neck while your phone is ringing by users demanding the functionality back (when already in production). Be ready to experiment and try out new things. Lastly, rely on FreeBSD to be a solid foundation in the storage space with its flexibility and options it provides for combining different solutions.

BENEDICT REUSCHLING is a documentation committer in the FreeBSD project and member of the documentation engineering team. He serves on the board of directors of the FreeBSD Foundation as vice president. 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.