SEE
TEXT
ONLY

pNFS

By Rick **Macklem**

A first attempt at creating a pNFS service for FreeBSD used GlusterFS along with the kernel-based "**nfsd**" communicating with it via fuse. It worked, but performance was abysmal with massive numbers of context switches. As such, I started over, designing an in-kernel service using the "**nfsd**" and some number of FreeBSD systems, but no cluster file system. This I refer to as **Plan B** and is discussed in this article.

Overall Goal

A pNFS service separates the Read/Write operations from all other NFSv4.1 operations. The hope is that this separation allows a pNFS service to be configured that exceeds the limits of a single NFS server for either storage capacity and/or I/O bandwidth. For a pNFS service, the NFS server becomes a Metadata Server (MDS) and handles all operations except for I/O operations. There are also other servers configured as data servers (DSs) that only handle I/O operations.

It is possible to configure mirroring within the DSs so that the data file for an MDS file will be mirrored on two or more of the DSs. When this is used, failure of a DS will not stop the pNFS service and a failed DS can be recovered once repaired while the pNFS service continues to operate. Although two-way mirroring would be the norm, it is possible to set a mirroring level of up to four or the number of DSs, whichever is less. The mirroring level refers to how many copies of the data file are kept on different DSs. The FreeBSD MDS is still a single point of failure, just as a normal NFS server is.

Overview of Plan B

A Plan B pNFS service consists of a single MDS and K DSs, all of which are FreeBSD12 systems. Clients will mount the MDS as they would a normal NFS server. When files are created, the MDS creates a file tree identical to what a normal NFS server creates, except that all the regular (VREG) files will be empty. As such, if you look at the exported tree on the MDS directly on the MDS server (not via an NFS mount), the files will all be of size 0. Each of these files will also have two extended attributes in the system attribute name space:

`pnfsd.dsfile` - This extended attribute stores the information that the MDS needs to find the data `file(s)` on `DS(s)` for this file.

`pnfsd.dsattr` - This extended attribute stores the Size, AccessTime, ModifyTime and Change attributes for the file, so that the MDS doesn't need to acquire the attributes from the DS for every `Getattr` operation.

For each regular (VREG) file, the MDS creates a data file on one or more, if mirroring is enabled, of the DSs in one of the `"dsN"` subdirectories. The name of this file is the file handle of the file on the MDS in hexadecimal so that the name is unique. The `DS(s)` are chosen in round-robin fashion when the file is created on the MDS and this seems to be adequate for storage of small files. For a service storing a mix of small and large files, a different algorithm may be needed.

I considered implementing a "most free space" algorithm, but have not done so, due to the overhead of checking how much free space each of the DSs has. The DSs use subdirectories named "`ds0`" to "`dsN`" so that no one directory gets too large. The value of "`N`" is set via the sysctl `vfs.nfsd.dsdirsize` on the MDS, with the default being 20.

For production servers that will store a lot of files, this value probably should be much larger. It can be increased when the "`nfsd`" daemon is not running on the MDS, once the additional "`dsN`" subdirectories are created on the DSs.

For pNFS aware `NFSv4.1` clients, the FreeBSD server will return two pieces of information to the client that allows it to do I/O directly to the DS.

• DeviceInfo - This is relatively static information that defines what a DS is. The critical information returned by the FreeBSD server is the IP address of the DS and, for the Flexible File layout, that it is "tightly coupled." There is a "`deviceid`" that identifies the DeviceInfo and which is used by the layout to reference it. The pNFS aware client acquires this information via the NFSv4.1 `GetDeviceInfo` operation.

• Layout - This is per file and can be recalled by the server when it is no longer valid. For the FreeBSD server, there is support for two types of layouts, called File and Flexible File layout respectively. Both allow the client to do I/O on the DS via `NFSv4.1` I/O operations. The Flexible File layout is a more recent variant that allows specification of mirrors, where the client is expected to do writes to all mirrors to maintain them in a consistent state. The Flexible File layout supports two variants referred to as "tightly coupled" vs "loosely coupled." The FreeBSD server always uses the "tightly coupled" variant, where the client uses the same credentials to do I/O on the DS as it would on the MDS. For the "loosely coupled" variant, the layout specifies a synthetic user/group that the client uses to do I/O on the DS. The FreeBSD server does not do striping and always returns layouts for the entire file. The critical information in a layout is Read vs Read/Write, the `deviceid(s)` that identify which DS(s) the data file is stored on and the file handle for the data file. The pNFS-aware client acquires this information via the `NFSv4.1 LayoutGet` operation. The client can also do a `LayoutReturn` operation to return the layout, either when it is done with it or when requested to do so by the NFSv4.1 server doing a `CBLayoutRecall` callback to the client. For Flexible File layouts, the client can report I/O errors when doing I/O on a DS to the MDS in the `LayoutReturn` arguments.

The MDS generates File layouts to `NFSv4.1` clients that know how to do pNFS for the non-mirrored DS case, unless the sysctl `vfs.nfsd.default_flexfile` is set non-zero, in which case Flexible File layouts are generated.

The mirrored DS configuration always generates Flexible File layouts. For NFS clients that do not support NFSv4.1 pNFS, all I/O operations are sent to the MDS. When the MDS receives an I/O RPC, it will do the RPC on the DS as a proxy.

If the DS is on the same machine as the MDS, the MDS/DS will do the RPC on the DS as a proxy and so on, until the machine runs out of some resource, such as session slots or mbufs. Therefore, a DS cannot be on the same system as the MDS.

Although I wouldn't consider it a practical production setup, for testing you can use a single system for all DSs and that system can also be used as the client, allowing testing using only two systems. For testing, it is possible to create more than one DS on the DS system, but you must assign an alias IP address to this DS system for each additional DS and mount the additional DSs using these separate alias addresses. In other words, only one DS mount per IP address is allowed. The mounts must also use separate exported directories for each DS in which to store data files.

The pNFS service is in FreeBSD-current and will be in FreeBSD12 when it is released. Prior to the FreeBSD12 release, it is possible to use a FreeBSD-current snapshot distribution for testing.

Setting up a FreeBSD pNFS Server Using Plan B

Let's do an example assuming five FreeBSD12 systems, with four of them configured as DSs, using two-way mirroring and `AUTH_SYS.`

• The MDS, exporting `/export` to the clients.
```
nfsv4-server
```

- The DSs with /DSstore exported to the MDS and clients for storage of data files.
  nfsv4-data0, nfsv4-data1, nfsv4-data2 and nfsv4-data3

On nfsv4-server, you will need to export a file system tree for the clients. The two lines in /etc/exports might look like:

```
V4: /export -sec=sys -network 192.168.1.0 -mask 255.255.255.0
/export -sec=sys -network 192.168.1.0 -mask 255.255.255.0
```

Then you will need the following lines in your /etc/rc.conf:

```
rpcbind_enable="YES"
mountd_enable="YES"
nfs_server_enable="YES"
nfsv4_server_enable="YES"
nfs_server_flags="-u -t -n 32 -p nfsv4-data0,nfsv4-data1,nfsv4-data2,
nfsv4-data3 -m 2"
```

Unless you wish to run the nfsuserd to map between uid/gid numbers and names, put these lines in /etc/sysctl.conf:

```
vfs.nfs.enable_uidtostring=1
vfs.nfsd.enable_stringtouid=1
```

This configures the NFSv4.1 server to use uid/gid numbers in the owner and owner_group strings.
Then the DSs must be mounted on the MDS. /etc/fstab lines like:

```
nfsv4-data0:/  /data0   nfsrw,nfsv4,minorversion=1,soft,retrans=2  0     0
nfsv4-data1:/  /data1   nfsrw,nfsv4,minorversion=1,soft,retrans=2  0     0
nfsv4-data2:/  /data2   nfsrw,nfsv4,minorversion=1,soft,retrans=2  0     0
nfsv4-data3:/  /data3   nfsrw,nfsv4,minorversion=1,soft,retrans=2  0     0
```

and the /data0, /data1, /data2 and /data3 directories will need to be created on the MDS for mounting of the DS data files. Note that "soft,retrans=2" would not normally be used for an NFSv4 mount, but this is an exception, since no state operations are done on the DS. Doing this allows the proxy operations to a DS to fail and cause the failing DS to be disabled when this occurs.
On the DSs, you will need to mkdir and export the /DSstore directory to the MDS and clients. The /etc/exports lines might look like:

```
V4: /DSstore -sec=sys -network 192.168.1.0 -mask 255.255.255.0
/DSstore -sec=sys -maproot=root nfsv4-server
/DSstore -sec=sys -network 192.168.1.0 -mask 255.255.255.0
```

You will need the following lines in your /etc/rc.conf:

```
rpcbind_enable="YES"
mountd_enable="YES"
nfs_server_enable="YES"
nfsv4_server_enable="YES"
nfs_server_flags="-u -t -n 32"
```

And in /etc/sysctl.conf:

```
vfs.nfs.enable_uidtostring=1
vfs.nfsd.enable_stringtouid=1
```

You will need to create the "dsN" directories under /DSstore. This command done in /DSstore on each of the DSs will do it: (All commands from here on will need to be done by root/su.)

```
  # jot -w ds 20 0 | xargs mkdir -m 700
```

Once these systems are set up, the MDS should be ready for client mounts. The FreeBSD clients will need the following in their `/etc/rc.conf` files:

```
rpcbind_enable="YES"
nfs_client_enable="YES"
nfscbd_enable="YES"
```

Then, on the FreeBSD client, the mount command might look like:

```
# mount -t nfs -o nfsv4,minorversion=1,pnfs nfsv4-server:/ /mnt
```

The client can then use "`/mnt`" as it would a normal NFS mount. If you do "`nfsstat -E -s`" on `nfsv4-server`, you should not see very many Read or Write operations. Most Read and Write operations should show up on a "`nfsstat -E -s`" done on the DSs. Having a few Read and Write operations on the MDS is normal, since that is what the clients fall back on when they fail to get a valid layout for any reason.

If, instead, you did a NFSv3 mount, the command might be:

```
# mount -t nfs nfsv4-server:/ /mnt
```

Now, if you do "`nfsstat -E -s`" on `nfsv4-server`, you will see a lot of Read and Write operations, since they are all being done through the MDS, which acts as a proxy for the DSs.

Let's suppose you create a file called "`abc.c`" on `/mnt` that is 274 bytes in size. Doing "`ls -l`" in `/mnt` would show a line like:

```
-rw-r--r--   1 ricktst  wheel  274 Jun  5 18:02 abc.c
```

Whereas if you go to `/export` on `nfsv4-server`, the "`ls -l`" line will look like:

```
-rw-r--r--   1 ricktst  wheel   0 Jun  5 18:02 abc.c
```

Then, a "`lsextattr system abc.c`" in the same directory will show:

```
abc.c pnfsd.dsfile     pnfsd.dsattr
```

and a "`pnfsdsfile abc.c`" will show:

```
abc.c: nfsv4-data2
ds5/207508569ff983350c000000a9730200eec58e800000000000000000
nfsv4-data3
ds5/207508569ff983350c000000a9730200eec58e800000000000000000
```

(The `pnfsdsfile` command shows what is in "`pnfsd.dsfile`" unless it has command line arguments specified.)

This tells you that the data for "`abc.c`" is stored on `nfsv4-data2` and `nfsv4-data3` in subdirectory "`ds5`" with filename "`2075...`".

If we go to `/DSstore/ds5` on `nfsv4-data2`, an "`ls -l *a97302*` will show:

```
-rw-r--r--  1 ricktst  wheel  274 Jun  5 18:02
207508569ff983350c000000a9730200eec58e800000000000000000
```

and on `nfsv4-data3`:

```
-rw-r--r--  1 ricktst  wheel  274 Jun  5 18:02
207508569ff983350c000000a9730200eec58e800000000000000000
```

Note that the ownership and permissions are the same as "`abc.c`" on the MDS. This is because it is a "tightly coupled" Flexible File layout service and enforces permission checking on the DS. For the Flexible File layout, the client can write to both `nfsv4-data2` and `nfsv4-data3` concurrently, so there should not be a significant performance hit caused by the mirroring. However, twice as much storage will be used as a non-mirrored configuration would use. The "`atime`" is not normally kept consistent across the DSs, but if the sysctl vfs.nfsd.pnfsstrictatime is set to one, it will be. Setting this does result in significant overheads.

Ok, so now let's have some fun with it. The MDS will disable a mirrored DS when one of three things occurs:

1. The MetaData Server (MDS) detects a problem when trying to do a proxy operation on the DS. This is why the DS servers are mounted on the MDS with the "`soft,retrans=2`" options. This can take a couple of minutes after the DS failure or network partitioning occurs.
2. A pNFS client can report an I/O error with respect to a DS to the MDS in the arguments for a LayoutReturn operation.
3. The system administrator can perform the `pnfsdskill(1)` command on the MDS to disable a DS. If the system administrator does a `pnfsdskill(1)` and it fails with `ENXIO` (Device not configured) that normally means the DS was already disabled via #1 or #2. Since doing this is harmless, once a system administrator knows that there is a problem with a mirrored DS, doing the command is recommended.

Let's do #3, since it is easy:

```
# pnfsdskill /data2
```

This will log a message on the console:

```
pNFS server: mirror nfsv4-data2 failed
```

`nfsv4-data2` has now been marked disabled and `CBLayoutRecall` callbacks have been done for all layouts using `nfsv4-data2`.

We can then unmount the `nfsv4-data2 /DSstore`:

```
# umount -N /data2
```

(The "`-N`" option ensures that the umount works even if threads are stuck trying to do RPCs on a failed `nfsv4-data2`.)

Now, let's write some data into `/mnt/abc.c`, so an "`ls -l`" line on the client looks like:

```
-rw-r--r--  1 ricktst  wheel  586 Jun  5 19:11 abc.c
```

If we go to `/DSstore/ds5` on `nfsv4-data2`, an "`ls -l *a97302*` will show:

```
-rw-r--r--  1 ricktst  wheel  274 Jun  5 18:02
207508569ff983350c000000a9730200eec58e800000000000000000
```

whereas on `nfsv4-data3`:

```
-rw-r--r--  1 ricktst  wheel  586 Jun  5 19:11
207508569ff983350c000000a9730200eec58e800000000000000000
```

Notice that `nfsv4-data2` didn't get written, since it is disabled. The file "`abc.c`" can still be read/written, but there is no longer a redundant copy.

The first step in repairing `nfsv4-data2` is to make sure that the out-of-date copy of the file on `nfsv4-data2` doesn't get used when `nfsv4-data2` is brought back online. To do this, we go to the `/export` directory on the MDS and replace `nfsv4-data2` with IP address `0.0.0.0` via the command:

```
# pnfsdsfile -r nfsv4-data2 abc.c
abc.c:    0.0.0.0
ds5/207508569ff983350c000000a9730200eec58e800000000000000000
nfsv4-data3.home.rick
ds5/207508569ff983350c000000a9730200eec58e800000000000000000
```

so, now it won't try to use `nfsv4-data2`.

This has to be done for all files in `/export` that specifies `nfsv4-data2` as a DS, so using `find(1)` the command is:

```
# find . -type f -exec pnfsdsfile -q -r nfsv4-data2 {} \;
```

Note that many files won't have `nfsv4-data2` specified as a DS, but the command knows to just `exit(0)` for these, so it can be safely used on any file.

Unfortunately, some combination of "`rename`" or "`link/unlink`" can result in `find(1)` missing some files. To check for missing ones, the command is:

```
# find . -type f -exec pnfsdsfile {} \; | sed "/nfsv4-data2/!d" | sed "s/:.*//"
```

(Using the "`sh`" shell to search for any files still specifying `nfsv4-data2`.)

The "`pnfsdsfile -r`" command needs to be done for any file names printed out by the above.

So, now we can safely bring `nfsv4-data2` back online after fixing it. To fix it for this exercise, we'll just go into `/DSstore` on `nfsv4-data2` and clean it up:

```
# cd /DSstore
# rm -rf *
# jot -w ds 20 0 | xargs mkdir -m 700
```

Now, on the MDS, we can bring it back online by mounting it and restarting the `nfsd` daemon:

```
# mount -t nfs -o nfsv4,minorversion=1,soft,retrans=2 nfsv4-data2:/ /data2
# /etc/rc.d/nfsd restart
```

After doing the above, newly created files may be assigned to `nfsv4-data2`, but the ones previously mirrored on `nfsv4-data2` still need to be recovered. To do this, we go to `/export` on the MDS and do the command:

```
# pnfsdscopymr -r /data2 abc.c
which copies the data for abc.c onto nfsv4-data2.
```

After doing this command, `pnfsdsfile(1)` shows:

```
# pnfsdsfile abc.c
abc.c:      nfsv4-data2.home.rick
ds5/207508569ff983350c000000a9730200eec58e800000000000000000
nfsv4-data3.home.rick
ds5/207508569ff983350c000000a9730200eec58e800000000000000000
```

If we go to `/DSstore/ds5` in `nfsv4-data2`, an "ls `-l *a97302*` will show:

```
-rw-r--r--  1 ricktst  wheel  586 Jun  5 19:11
207508569ff983350c000000a9730200eec58e800000000000000000
```

and on `nfsv4-data3`:

```
-rw-r--r--  1 ricktst  wheel  586 Jun  5 19:11
207508569ff983350c000000a9730200eec58e800000000000000000
```

The code that implements this copy in the kernel is somewhat involved. A brief description of the algorithm is:

- The MDS file's `vnode` is locked, blocking `LayoutGet` operations.
- Disable issuing of `Read/Write` layouts for the file via the `nfsdontlist`, so that they will be disabled after the MDS file's vnode is unlocked.
- Set up the `nfsrv_recalllist` so that recall of `read/write` layouts can be done.
- Unlock the MDS file's vnode, so that the `client(s)` can perform proxied writes, `LayoutCommits` and `LayoutReturns` for the file when completing the `LayoutReturn` requested by the `LayoutRecall` callback.
- Issue a `CBLayoutRecall` callback for all `Read/Write` layouts and wait for them to be returned. (If the `CBLayoutRecall` callback replies `NFSERR_NOMATCHLAYOUT`, they are gone and no `LayoutReturn` is needed.)
- Exclusively lock the MDS file's vnode. This ensures that no proxied writes are in progress or can occur during the DS file copy.

It also blocks `Setattr` operations.
- Create the file on the repaired mirror.
- Copy the file from the operational DS.
- Copy any ACL from the MDS file to the new DS file.
- Set the modify time of the new DS file to that of the MDS file.
- Update the extended attribute for the MDS file.
- Enable issuing of `Read/Write` layouts by deleting the `nfsdontlist` entry.
- Unlock the MDS file's vnode allowing operations to continue normally, since it is again mirrored.

Again, this has to be done for all files, so the `find(1)` command is:

```
# find . -type f -exec pnfsdscopymr -r /data2 {} \;
```

and to check for ones missed by the `find(1)`:

```
# find . -type f -exec pnfsdsfile {} \; | sed "/0\.0\.0\.0/!d" | sed "/:.*//"
```

(Using `"sh"`, search for any files that still have a DS address of `0.0.0.0`.)

If this prints `file(s)`, the "`pnfsdscopymr -r`" command needs to be done on them. If nothing gets printed out, `nfsv4-data2` has been recovered and all files should now be mirrored correctly.

A system administrator can also use the `pnfsdscopymr(1)` command to migrate the data file from one DS to another DS. To move the data file for `abc.c` from `nfsv4-data3` to `nfsv4-data0`, the command is:

```
# pnfsdscopymr -m /data3 /data0 abc.c
```

After this command, `pnfsdsfile` shows:

```
# pnfsdsfile abc.c
abc.c:      nfsv4-data2.home.rick
ds5/207508569ff983350c000000a9730200eec58e800000000000000000
nfsv4-data0.home.rick
ds5/207508569ff983350c000000a9730200eec58e800000000000000000
```

If we go to `/DSstore/ds5` in `nfsv4-data2`, an "`ls -l *a97302*` will show:

```
-rw-r--r--  1 ricktst  wheel  586 Jun  5 19:11
207508569ff983350c000000a9730200eec58e800000000000000000
```

on `nfsv4-data3`:

```
ls: No match.
```

and on `nfsv4-data0`:

```
-rw-r--r--  1 ricktst  wheel  586 Jun  5 19:11
207508569ff983350c000000a9730200eec58e800000000000000000
```

This might be useful to move the data for a large file to a DS with more free space.

The "`pnfsdscopymr -m`" command just does some sanity checking followed by one system call to do the work. This system call could be used by a storage/load balancer implementation in the future.

The Linux client currently has a Flexible File layout driver that supports the "loosely coupled" variant but does not handle the "tightly coupled" variant correctly. It always uses the synthetic user/group for I/O operations on a DS.

There are two ways to deal with this:
1. Patch the client driver to fix this. I have a patch here:
   http://people.freebsd.org/~rmacklem/flexfile.patch which seems to work ok.
2. Export `/DSstore` on the DSs with "`-maproot=root`" and then set the `vfs.nfsd.flexlinuxhack=1` on the MDS. Setting this sysctl makes the pNFS server send the synthetic/group of (`0, 0`) in the layout, so that the Linux client always does I/O on the DSs as "`root`".

You also need a recent Linux kernel. I have been testing with Linux-4.17-rc2. Linux-4.12 worked but would crash intermittently during testing and earlier kernels don't have Flexible File layout support for NFSv4.1 DSs. This is not a problem for the non-mirrored pNFS server, since it will send File layouts to the Linux client.

Once you have resolved this, the mount command on Linux is:

```
# mount -t nfs -o nfsvers=4,minorversion=1 nfs4-server:/ /mnt
```

For more information on the setup and management of the pNFS service, the document http://people.freebsd.org/~rmacklem/pnfs-planb-setup.txt might be useful. There are also man pages for `pnfsdskill(1)`, `pnfsdsfile(1)`, `pnfsdscopymr(1)` and `pnfs(4)`.

If you wish to look at the protocol details, the RFCs are:
RFC-5661: Network File System (NFS) Version 4 Minor Version 1 Protocol, ISSN: 2070-1721.
The flexible file layout is currently an internet draft but should be published as an RFC soon. Hopefully before this article is published. If not, here is the draft:
Parallel NFS (pNFS) Flexible File Layout `draft-ietf-nfsv4-flex-files-19.txt`.

Rick Macklem has been working with BSD for way too long. His first contribution to BSD was a port of 4.2BSD to the MicroVAXII in 1985. Shortly after that, he contributed the first NFS implementation that became part of 4.3BSD Reno. He worked as a sysadmin for a Canadian university for thirty years and is now happily retired and still working on NFS for FreeBSD. And, yes, this article was written using "ed", which is still his editor of choice.