Published on **The O'Reilly Network** (http://www.oreillynet.com/)
http://www.oreillynet.com/pub/a/bsd/2002/02/14/Big_Scary_Daemons.html
See this if you're having trouble printing code examples

Understanding NFS

02/14/2002

We've discussed sharing filesystems via SMB a few times. SMB lets you access files shared by a Windows system after jumping through only half a dozen loops. Sharing files with another Unix system is much, much simpler. FreeBSD supports the Unix standard Network File System out of the box. NFS intimidates many junior system administrators, but it's really quite simple once you know what's going on.

Each NFS connection works on a client-server model. One computer is the server and offers filesystems to other systems. This is called "NFS exporting," and the filesystems offered are called "exports." The clients can mount server exports in a manner almost identical to that used to mount local filesystems.

One interesting thing about NFS is it's statelessness. You can reboot a server and the client won't crash. It won't be able to access files on the server's export while the server is down, but once it returns, you'll pick up right where things left off. Other network file sharing systems are not so resilient.

In this discussion, we're going to link multiple FreeBSD to 4-STABLE systems. Each NFS implementation is slightly different. You will find minor NFS variations between Solaris, Linux, BSD, and other Unixes. NFS should work between them all, but it might require the occasional tweak. If you have problems interoperating with some other Unix, check the freebsd-net mailing list archive; the issue has almost certainly been discussed there.

Let's configure some systems for NFS. Both server and clients require NFS support in the kernel, but the various NFS commands dynamically load the appropriate module into the kernel. FreeBSD's generic kernel supports NFS, but if you customize your kernel and don't like loading file system support as a module, be sure your kernel configuration includes:

```
options         NFS
```

First of all, we have the server side. You can enable basic NFS exports with the following rc.conf options:

```
portmap_enable="YES"
nfs_server_enable="YES"
```

Portmap provides, as you might guess, a mapping service for network ports. Different exports and clients require unique network ports. Clients ask portmap which port they should connect to for their actual mount. The `nfs_server_enable` option starts nfsd and mountd. mountd just listens for incoming NFS requests on assorted high-numbered network ports, and makes these port numbers

available to portmap. When clients talk to portmap and mountd, nfsd actually handles their requests.

Once you reboot, your server should show something like the following amongst it's `sockstat`(1) output. This shows that server-side NFS is running more or less properly. If you don't see something resembling this, check `/var/log/messages` for log messages indicating your problem.

```
# sockstat -4
USER    COMMAND   PID  FD PROTO  LOCAL ADDRESS  FOREIGN ADDRESS
root    rpc.stat   77   3 udp4   *:1011         *:*
root    rpc.stat   77   4 tcp4   *:1022         *:*
root    nfsd       70   3 tcp4   *:2049         *:*
root    mountd     68   3 udp4   *:1023         *:*
root    mountd     68   5 tcp4   *:1023         *:*
daemon  portmap    66   3 udp4   *:111          *:*
daemon  portmap    66   4 tcp4   *:111          *:*
```

Your client won't show any special `sockstat` output before network shares are mounted, but a `ps -ax` will display several nfsiod processes.

Now that your systems are prepared to handle NFS, we need to tell the server which directories it can export. We could just export the entire server, but that's not generally a good thing. Clients should have little or no need to remote-mount the server's root filesystem, for example. Define allowed exports in the `/etc/exports` file. This file has a separate line for each hard-drive partition on the server. Each line has up to three components:

- the directories to be exported
- the options on that export
- the clients that can connect

Each combination of clients and server disk partition can only have one line in the exports file. This means that if `/usr/ports` and `/usr/home` are on the same partition, they must be exported in the same line to any one client. You don't have to export the entire partition, mind you; you can just as easily share out a single directory within a partition. This directory must be a true path, and it must not contain either symlinks or double or single dots (i.e., ".."). If I wanted to export my home directory to every host on the Internet, I could use an `/etc/exports` line that consisted entirely of this:

```
/usr/home/mwlucas
```

We have no options and no host restrictions. Now that we've edited the exports file, we have to tell mountd to re-read it.

```
killall -1 mountd
```

Any problems will appear in `/var/log/messages`. For example, when I was first testing this, I had a single entry in `/etc/exports` of `/home/mwlucas`. It was about this time that I learned that `/etc/exports` cannot contain symlinks. FreeBSD puts user home directories under `/usr/home` and uses a symlink to create the `/home` directory. The error log gave me a warning like this:

```
Jan 24 07:13:35 server mountd[68]: bad exports list line /home/mwlucas
```

This told me where the problem was; identifying the problem was my job.

Now, over on the client side, I create the directory `$HOME/serverhome`. I want to NFS-mount my home directory on the server onto this directory. This looks almost exactly like a standard mount(8) command. mount takes two arguments: the physical device you're using and the mount point. In this case, our physical device is a remote server and the exported filesystem:

```
# mount server:/usr/home/mwlucas serverhome
#
```

Once this finishes, test your mount with `df`(1):

```
# df
Filesystem                1K-blocks     Used    Avail Capacity  Mounted on
/dev/ad0s1a                   99183    67411    23838    74%    /
/dev/ad0s1f                 5186362  3873110   898344    81%    /usr
/dev/ad0s1e                  198399    21211   161317    12%    /var
procfs                            4        4        0   100%    /proc
server:/usr/home/mwlucas  34723447  3886523 28059049    12%
 /usr/home/mwlucas/serverhome
#
```

One thing to note is that NFS uses the same usernames on each side of the connection. My files are owned by mwlucas on the server, so they are owned by mwlucas on the client. This can be a problem on a large network where users have root on their own machines. To create a central repository of authorized users, consider Kerberos or NIS. On a small network, or on a network with limited administrators, this usually isn't a problem.

Looking at the contents of `$HOME/serverhome`, we see that the files are mostly owned by mwlucas, with the occasional file owned by root. Those files shouldn't be there, really; I have some root-owned .png files that are screenshots of terminal windows that I took for my book. Since I happen to be thinking about it, I'll remove them.

```
# su
# cd serverhome
# rm usersetup.png
override rw-r--r--  root/mwlucas for usersetup.png? y
rm: usersetup.png: Permission denied
#
```

But I'm root! Why would it not let me delete a file?

I'm root on the client, but not on the server. The server doesn't trust root on other machines to execute commands as root on the server. It does trust usernames, however. NFS has a special option for handling root; you can map requests from root to any other username. For example, you might say that all requests from "root" on a client will run as "nfsroot" on the server. With careful use of groups, you could allow this nfsroot user to have limited access to things. Use the `-maproot` option to map root to another user.

This is my home network and I don't have other users, however, so I can just trust the client. In this case, I map NFS root requests to uid 0, aka root. Here's the new exports file.

```
/usr/home/mwlucas -maproot=0
```

I have to restart mountd again. Since NFS is stateless, old mounts remain in effect. On the client, I'm even in the same directory; restarting mountd didn't affect my client at all. The `rm` runs flawlessly now.

So, what if I want to export another directory on the same partition? For example, suppose I want to export `/usr/src` to my laptop to save space on the hard drive? List the other directories in `/etc/exports`, right after the first exported directory, separated by a space. While I'm at it, I could also NFS-mount `/usr/obj`. This way, I could run `make buildworld` on my fast machine and run `make installworld` on my slow laptop, greatly accelerating upgrades. My `/etc/exports` now looks like this:

```
/usr/home/mwlucas /usr/src /usr/obj -maproot=0
```

There are no identifiers between the components of the line. Yes, it would be easier to read if we could put each shared directory on its own line, but we can't; they're all on the same partition. The FreeBSD team could rewrite this so that it had more structure, but then our `/etc/exports` would be incompatible with that from any other Unix. Restart mountd and mount these filesytems:

```
# mount server:/usr/obj /usr/obj
# mount server:/usr/src /usr/src
```

My client filesystems now look like this:

```
# df
Filesystem       1K-blocks     Used    Avail Capacity  Mounted on
/dev/ad0s1a          99183    67411    23838    74%    /
/dev/ad0s1f        5186362  3873110   898344    81%    /usr
/dev/ad0s1e         198399    21212   161316    12%    /var
procfs                   4        4        0   100%    /proc
server:/usr/obj   34723447  3886485 28059087    12%    /usr/obj
server:/usr/src   34723447  3886485 28059087    12%    /usr/src
#
```

A `buildworld` on my laptop takes two hours; on the server, it takes thirty minutes. I ran a `make buildworld` and a `make buildkernel` earlier on the server; if I wanted the same kernel, I could just go into `/usr/src` and type `make installkernel && make installworld && mergemaster`, and the FreeBSD built on my server would be installed on the laptop.

To make things slightly more difficult, my home system is on a cable modem. I'm using one of FreeBSD's built-in firewalls to protect the system, but it would be nice to add some security into NFS itself. You can easily restrict NFS mounts by IP address, allowing only certain clients to mount an exported share. Just specify the host name or IP address at the end of the partition's `/etc/exports` entry. My client has an IP address of 192.168.1.200, so my `/etc/exports` now looks like this:

```
/usr/home/mwlucas /usr/src /usr/obj -maproot=0 192.168.1.200
```

This quickly becomes very convenient for upgrades. What's nice is that you can also build ports on the server, and install them on the client, by exporting `/usr/ports`. This is starting to get a little ridiculous, however; eventually, `/etc/exports` will have entries for almost every directory in `/usr`. I might as well export the whole of `/usr` and get it over with. This isn't that great, either, because I want to mount the exported directories at different places. The exported ports tree should go over `/usr/ports`, for example, but I don't want my server's home directory overwriting mine. Fortunately, there's an easy solution. The `-alldirs` option allows you to export a partition, and all the directories beneath it. You must specify a partition when you use `alldirs`. Multiple options are separated by a comma.

```
/usr -alldirs,maproot=0 192.168.1.200
```

Now I hup mountd again, and all of `/usr` can be exported and mounted separately. For example, I still had `/usr/src` and `/usr/obj` NFS-mounted. Quick tests with `df` and `ls` show that they're still there, still accessible, and still serving files, even though I've completely redone the way they're exported. I can mount arbitrary directories from the server's `/usr`.

Finally, here's a tip on NFS performance. By default, FreeBSD uses conservative NFS mounting options. These work well when trying to interoperate with other Unixes; everybody speaks the lowest common denominator. You can use mount options to augment NFS performance but reduce interoperability somewhat. These options aren't necessary when you're working with one or two clients, but as your NFS installation grows, you'll find them helpful. They may or may not work with other operating systems; it depends on what those OSs support.

First of all, NFS runs over UDP by default. The `tcp` option tells the client to request a mount over TCP.

Then we want to make the mount interruptible. This means that if the server goes away, for any reason, client programs that are trying to access the export can be interrupted. Otherwise, the client will continue trying to access the export until the filesystem times out. Set this with the `intr` option.

NFS comes in many versions. The latest one in wide use is Version 3. You can request this with the `nfsv3` option.

Finally, you have the size of read and write requests. The defaults are rather small, being well suited to networks of the early nineties. You can set the read and write size to more modern values with the `-r` and `-w` options. 32768 is a good value for both.

So, putting this all together, I want to mount my exports in the following way.

```
# mount -o tcp,intr,nfsv3,-w=32768,-r=32768 server:/usr/home/mwlucas serverhome
```

This will give me close-to-optimal performance with minimal effort. Further fine-tuning NFS requires testing various options with your particular network equipment.

Now that I have NFS running at home, I find myself mounting from one machine to others for just about any reason. It's well worth the minor trouble to set up.

Michael Lucas lives in a haunted house in Detroit, Michigan with his wife Liz, assorted rodents,

and a multitude of fish. He's the network architect for the Great Lakes Technologies Group, which is simply a nice way of saying it's all his problem.

---

*Read more from **Big Scary Daemons**.*

Return to the BSD DevCenter.