Linux NFS-HOWTO

Tavis Barr

tavis dot barr at liu dot edu

Nicolai Langfeldt

janl at linpro dot no

Seth Vidal

skvidal at phy dot duke dot edu

Tom McNeal

trmcneal at attbi dot com

2002-08-25

Revision History

Revision v3.1 2002-08-25 Revised by: tavis
Typo in firewalling section in 3.0
Revision v3.0 2002-07-16 Revised by: tavis
Updates plus additions to performance, security

1. Preamble

1.1. Legal stuff

1.2. Disclaimer

This document is provided without any guarantees, including merchantability or fitness for a particular use. The maintainers cannot be responsible if following instructions in this document leads to damaged equipment or data, angry neighbors, strange habits, divorce, or any other calamity.

1.3. Feedback

This will never be a finished document; we welcome feedback about how it can be improved. As of February 2002, the Linux NFS home page is being hosted at http://nfs.sourceforge.net. Check there for mailing lists, bug fixes, and updates, and also to verify who currently maintains this document.

1.4. Translation

If you are able to translate this document into another language, we would be grateful and we will also do our best to assist you. Please notify the maintainers.

1.5. Dedication

NFS on Linux was made possible by a collaborative effort of many people, but a few stand out for special recognition. The original version was developed by Olaf Kirch and Alan Cox. The version 3 server code was solidified by Neil Brown, based on work from Saadia Khan, James Yarbrough, Allen Morris, H.J. Lu, and others (including himself). The client code was written by Olaf Kirch and updated by Trond Myklebust. The version 4 lock manager was developed by Saadia Khan. Dave Higgen and H.J. Lu both have undertaken the thankless job of extensive maintenance and bug fixes to get the code to actually work the way it was supposed to. H.J. has also done extensive development of the nfs-utils package. Of course this dedication is leaving many people out.

The original version of this document was developed by Nicolai Langfeldt. It was heavily rewritten in 2000 by Tavis Barr and Seth Vidal to reflect substantial changes in the workings of NFS for Linux developed between the 2.0 and 2.4 kernels. It was edited again in February 2002, when Tom McNeal made substantial additions to the performance section. Thomas Emmel, Neil Brown, Trond Myklebust, Erez Zadok, and Ion Badulescu also provided valuable comments and contributions.

2. Introduction

2.1. What is NFS?

The Network File System (NFS) was developed to allow machines to mount a disk partition on a remote machine as if it were on a local hard drive. This allows for fast, seamless sharing of files across a network.

It also gives the potential for unwanted people to access your hard drive over the network (and thereby possibly read your email and delete all your files as well as break into your system) if you set it up incorrectly. So please read the Security section of this document carefully if you intend to implement an NFS setup.

There are other systems that provide similar functionality to NFS. Samba (http://www.samba.org) provides file services to Windows clients. The Andrew File System from IBM (http://www.transarc.com/Product/EFS/AFS/index.html), recently open-sourced, provides a file sharing mechanism with some additional security and performance features. The Coda File System (http://www.coda.cs.cmu.edu/) is still in development as of this writing but is designed to work well with disconnected clients. Many of the features of the Andrew and Coda file systems are slated for inclusion in the next version of NFS (Version 4) (http://www.nfsv4.org). The advantage of NFS today is that it is mature, standard, well understood, and supported robustly across a variety of platforms.

2.2. What is this HOWTO and what is it not?

This HOWTO is intended as a complete, step-by-step guide to setting up NFS correctly and effectively. Setting up NFS involves two steps, namely configuring the server and then configuring the client. Each of these steps is dealt with in order. The document then offers some tips for people with particular needs and hardware setups, as well as security and troubleshooting advice.

This HOWTO is not a description of the guts and underlying structure of NFS. For that you may wish to read *Linux NFS and Automounter Administration* by Erez Zadok (Sybex, 2001). The classic NFS book, updated and still quite useful, is *Managing NFS and NIS* by Hal Stern, published by O'Reilly & Associates, Inc. A much more advanced technical description of NFS is available in *NFS Illustrated* by Brent Callaghan.

This document is also not intended as a complete reference manual, and does not contain an exhaustive list of the features of Linux NFS. For that, you can look at the man pages for *nfs(5)*, *exports(5)*, *mount(8)*, *fstab(5)*, *nfsd(8)*, *lockd(8)*, *statd(8)*, *rquotad(8)*, and *mountd(8)*.

It will also not cover PC-NFS, which is considered obsolete (users are encouraged to use Samba to share files with Windows machines) or NFS Version 4, which is still in development.

2.3. Knowledge Pre-Requisites

You should know some basic things about TCP/IP networking before reading this HOWTO; if you are in doubt, read the Networking- Overview-HOWTO (http://www.linuxdoc.org/HOWTO/Networking-Overview-HOWTO.html).

2.4. Software Pre-Requisites: Kernel Version and nfs-utils

The difference between Version 2 NFS and version 3 NFS will be explained later on; for now, you might simply take the suggestion that you will need NFS Version 3 if you are installing a dedicated or high-volume file server. NFS Version 2 should be fine for casual use.

NFS Version 2 has been around for quite some time now (at least since the 1.2 kernel series) however you will need a kernel version of at least 2.2.18 if you wish to do any of the following:

- Mix Linux NFS with other operating systems' NFS
- Use file locking reliably over NFS
- Use NFS Version 3.

There are also patches available for kernel versions above 2.2.14 that provide the above functionality. Some of them can be downloaded from the Linux NFS homepage. If your kernel version is 2.2.14-2.2.17 and you have the source code on hand, you can tell if these patches have been added because NFS Version 3 server support will be a configuration option. However, unless you have some particular reason to use an older kernel, you should upgrade because many bugs have been fixed along the way. Kernel 2.2.19 contains some additional locking improvements over 2.2.18.

Version 3 functionality will also require the nfs-utils package of at least version 0.1.6, and mount version 2.10m or newer. However because nfs-utils and mount are fully backwards compatible, and because newer versions have lots of security and bug fixes, there is no good reason not to install the newest nfs-utils and mount packages if you are beginning an NFS setup.

All 2.4 and higher kernels have full NFS Version 3 functionality.

In all cases, if you are building your own kernel, you will need to select NFS and NFS Version 3 support at compile time. Most (but not all) standard distributions come with kernels that support NFS version 3.

Handling files larger than 2 GB will require a 2.4x kernel and a 2.2.x version of glibc.

All kernels after 2.2.18 support NFS over TCP on the client side. As of this writing, server-side NFS over TCP only exists in a buggy form as an experimental option in the post-2.2.18 series; patches for 2.4 and

2.5 kernels have been introduced starting with 2.4.17 and 2.5.6. The patches are believed to be stable, though as of this writing they are relatively new and have not seen widespread use or integration into the mainstream 2.4 kernel.

Because so many of the above functionalities were introduced in kernel version 2.2.18, this document was written to be consistent with kernels above this version (including 2.4.x). If you have an older kernel, this document may not describe your NFS system correctly.

As we write this document, NFS version 4 has only recently been finalized as a protocol, and no implementations are considered production-ready. It will not be dealt with here.

2.5. Where to get help and further information

As of November 2000, the Linux NFS homepage is at http://nfs.sourceforge.net. Please check there for NFS related mailing lists as well as the latest version of nfs-utils, NFS kernel patches, and other NFS related packages.

When you encounter a problem or have a question not covered in this manual, the faq or the man pages, you should send a message to the nfs mailing list (<`nfs@lists.sourceforge.net`>). To best help the developers and other users help you assess your problem you should include:

- the version of nfs-utils you are using
- the version of the kernel and any non-stock applied kernels.
- the distribution of linux you are using
- the version(s) of other operating systems involved.

It is also useful to know the networking configuration connecting the hosts.

If your problem involves the inability mount or export shares please also include:

- a copy of your `/etc/exports` file
- the output of **rpcinfo -p** *localhost* run on the server
- the output of **rpcinfo -p** *servername* run on the client

Sending all of this information with a specific question, after reading all the documentation, is the best way to ensure a helpful response from the list.

You may also wish to look at the man pages for *nfs(5)*, *exports(5)*, *mount(8)*, *fstab(5)*, *nfsd(8)*, *lockd(8)*, *statd(8)*, *rquotad(8)*, and *mountd(8)*.

3. Setting Up an NFS Server

3.1. Introduction to the server setup

It is assumed that you will be setting up both a server and a client. If you are just setting up a client to work off of somebody else's server (say in your department), you can skip to Section 4. However, every client that is set up requires modifications on the server to authorize that client (unless the server setup is done in a very insecure way), so even if you are not setting up a server you may wish to read this section to get an idea what kinds of authorization problems to look out for.

Setting up the server will be done in two steps: Setting up the configuration files for NFS, and then starting the NFS services.

3.2. Setting up the Configuration Files

There are three main configuration files you will need to edit to set up an NFS server: `/etc/exports`, `/etc/hosts.allow`, and `/etc/hosts.deny`. Strictly speaking, you only need to edit `/etc/exports` to get NFS to work, but you would be left with an extremely insecure setup. You may also need to edit your startup scripts; see Section 3.3.3 for more on that.

3.2.1. /etc/exports

This file contains a list of entries; each entry indicates a volume that is shared and how it is shared. Check the man pages (**man exports**) for a complete description of all the setup options for the file, although the description here will probably satistfy most people's needs.

An entry in `/etc/exports` will typically look like this:

```
 directory machine1(option11,option12) machine2(option21,option22)
```

where

directory

the directory that you want to share. It may be an entire volume though it need not be. If you share a directory, then all directories under it within the same file system will be shared as well.

machine1 and machine2

client machines that will have access to the directory. The machines may be listed by their DNS address or their IP address (e.g., *machine.company.com* or *192.168.0.8*). Using IP addresses is more reliable and more secure. If you need to use DNS addresses, and they do not seem to be resolving to the right machine, see Section 7.3.

optionxx

the option listing for each machine will describe what kind of access that machine will have. Important options are:

- **`ro`**: The directory is shared read only; the client machine will not be able to write to it. This is the default.
- **`rw`**: The client machine will have read and write access to the directory.
- **`no_root_squash`**: By default, any file request made by user `root` on the client machine is treated as if it is made by user `nobody` on the server. (Excatly which UID the request is mapped to depends on the UID of user "nobody" on the server, not the client.) If **`no_root_squash`** is selected, then root on the client machine will have the same level of access to the files on the system as root on the server. This can have serious security implications, although it may be necessary if you want to perform any administrative work on the client machine that involves the exported directories. You should not specify this option without a good reason.
- **`no_subtree_check`**: If only part of a volume is exported, a routine called subtree checking verifies that a file that is requested from the client is in the appropriate part of the volume. If the entire volume is exported, disabling this check will speed up transfers.
- **`sync`**: By default, all but the most recent version (version 1.11) of the **exportfs** command will use **`async`** behavior, telling a client machine that a file write is complete - that is, has been written to stable storage - when NFS has finished handing the write over to the filesysytem. This behavior may cause data corruption if the server reboots, and the **`sync`** option prevents this. See Section 5.9 for a complete discussion of **`sync`** and **`async`** behavior.

Suppose we have two client machines, *slave1* and *slave2*, that have IP addresses *192.168.0.1* and *192.168.0.2*, respectively. We wish to share our software binaries and home directories with these machines. A typical setup for `/etc/exports` might look like this:

```
/usr/local   192.168.0.1(ro) 192.168.0.2(ro)
/home        192.168.0.1(rw) 192.168.0.2(rw)
```

Here we are sharing `/usr/local` read-only to slave1 and slave2, because it probably contains our software and there may not be benefits to allowing slave1 and slave2 to write to it that outweigh security concerns. On the other hand, home directories need to be exported read-write if users are to save work on them.

If you have a large installation, you may find that you have a bunch of computers all on the same local network that require access to your server. There are a few ways of simplifying references to large numbers of machines. First, you can give access to a range of machines at once by specifying a network and a netmask. For example, if you wanted to allow access to all the machines with IP addresses between *192.168.0.0* and *192.168.0.255* then you could have the entries:

```
/usr/local 192.168.0.0/255.255.255.0(ro)
/home      192.168.0.0/255.255.255.0(rw)
```

See the Networking-Overview HOWTO (http://www.linuxdoc.org/HOWTO/Networking-Overview-HOWTO.html) for further information about how netmasks work, and you may also wish to look at the man pages for `init` and `hosts.allow`.

Second, you can use NIS netgroups in your entry. To specify a netgroup in your exports file, simply prepend the name of the netgroup with an "@". See the NIS HOWTO (http://www.linuxdoc.org/HOWTO/NIS-HOWTO.html) for details on how netgroups work.

Third, you can use wildcards such as **.foo.com* or *192.168.* instead of hostnames. There were problems with wildcard implementation in the 2.2 kernel series that were fixed in kernel 2.2.19.

However, you should keep in mind that any of these simplifications could cause a security risk if there are machines in your netgroup or local network that you do not trust completely.

A few cautions are in order about what cannot (or should not) be exported. First, if a directory is exported, its parent and child directories cannot be exported if they are in the same filesystem. However, exporting both should not be necessary because listing the parent directory in the `/etc/exports` file will cause all underlying directories within that file system to be exported.

Second, it is a poor idea to export a FAT or VFAT (i.e., MS-DOS or Windows 95/98) filesystem with NFS. FAT is not designed for use on a multi-user machine, and as a result, operations that depend on permissions will not work well. Moreover, some of the underlying filesystem design is reported to work poorly with NFS's expectations.

Third, device or other special files may not export correctly to non-Linux clients. See Section 8 for details on particular operating systems.

3.2.2. /etc/hosts.allow and /etc/hosts.deny

These two files specify which computers on the network can use services on your machine. Each line of the file contains a single entry listing a service and a set of machines. When the server gets a request from a machine, it does the following:

- It first checks `hosts.allow` to see if the machine matches a description listed in there. If it does, then the machine is allowed access.
- If the machine does not match an entry in `hosts.allow`, the server then checks `hosts.deny` to see if the client matches a listing in there. If it does then the machine is denied access.
- If the client matches no listings in either file, then it is allowed access.

In addition to controlling access to services handled by **inetd** (such as telnet and FTP), this file can also control access to NFS by restricting connections to the daemons that provide NFS services. Restrictions are done on a per-service basis.

The first daemon to restrict access to is the portmapper. This daemon essentially just tells requesting clients how to find all the NFS services on the system. Restricting access to the portmapper is the best defense against someone breaking into your system through NFS because completely unauthorized clients won't know where to find the NFS daemons. However, there are two things to watch out for. First, restricting portmapper isn't enough if the intruder already knows for some reason how to find those daemons. And second, if you are running NIS, restricting portmapper will also restrict requests to NIS. That should usually be harmless since you usually want to restrict NFS and NIS in a similar way, but just be cautioned. (Running NIS is generally a good idea if you are running NFS, because the client machines need a way of knowing who owns what files on the exported volumes. Of course there are other ways of doing this such as syncing password files. See the NIS HOWTO (http://www.linuxdoc.org/HOWTO/NIS-HOWTO.html) for information on setting up NIS.)

In general it is a good idea with NFS (as with most internet services) to explicitly deny access to IP addresses that you don't need to allow access to.

The first step in doing this is to add the followng entry to `/etc/hosts.deny`:

```
portmap:ALL
```

Starting with nfs-utils 0.2.0, you can be a bit more careful by controlling access to individual daemons. It's a good precaution since an intruder will often be able to weasel around the portmapper. If you have a newer version of nfs-utils, add entries for each of the NFS daemons (see the next section to find out what these daemons are; for now just put entries for them in hosts.deny):

```
    lockd:ALL
    mountd:ALL
    rquotad:ALL
    statd:ALL
```

Even if you have an older version of nfs-utils, adding these entries is at worst harmless (since they will just be ignored) and at best will save you some trouble when you upgrade. Some sys admins choose to put the entry **`ALL:ALL`** in the file `/etc/hosts.deny`, which causes any service that looks at these files to deny access to all hosts unless it is explicitly allowed. While this is more secure behavior, it may also get you in trouble when you are installing new services, you forget you put it there, and you can't figure out for the life of you why they won't work.

Next, we need to add an entry to `hosts.allow` to give any hosts access that we want to have access. (If we just leave the above lines in `hosts.deny` then nobody will have access to NFS.) Entries in `hosts.allow` follow the format

```
    service: host [or network/netmask] , host [or network/netmask]
```

Here, host is IP address of a potential client; it may be possible in some versions to use the DNS name of the host, but it is strongly discouraged.

Suppose we have the setup above and we just want to allow access to *slave1.foo.com* and *slave2.foo.com*, and suppose that the IP addresses of these machines are *192.168.0.1* and *192.168.0.2*, respectively. We could add the following entry to `/etc/hosts.allow`:

```
   portmap: 192.168.0.1 , 192.168.0.2
```

For recent nfs-utils versions, we would also add the following (again, these entries are harmless even if they are not supported):

```
    lockd: 192.168.0.1 , 192.168.0.2
    rquotad: 192.168.0.1 , 192.168.0.2
    mountd: 192.168.0.1 , 192.168.0.2
    statd: 192.168.0.1 , 192.168.0.2
```

If you intend to run NFS on a large number of machines in a local network, `/etc/hosts.allow` also allows for network/netmask style entries in the same manner as `/etc/exports` above.

3.3. Getting the services started

3.3.1. Pre-requisites

The NFS server should now be configured and we can start it running. First, you will need to have the appropriate packages installed. This consists mainly of a new enough kernel and a new enough version of the nfs-utils package. See Section 2.4 if you are in doubt.

Next, before you can start NFS, you will need to have TCP/IP networking functioning correctly on your machine. If you can use telnet, FTP, and so on, then chances are your TCP networking is fine.

That said, with most recent Linux distributions you may be able to get NFS up and running simply by rebooting your machine, and the startup scripts should detect that you have set up your `/etc/exports` file and will start up NFS correctly. If you try this, see Section 3.4 Verifying that NFS is running. If this does not work, or if you are not in a position to reboot your machine, then the following section will tell you which daemons need to be started in order to run NFS services. If for some reason **nfsd** was already running when you edited your configuration files above, you will have to flush your configuration; see Section 3.5 for details.

3.3.2. Starting the Portmapper

NFS depends on the portmapper daemon, either called **portmap** or **rpc.portmap**. It will need to be started first. It should be located in `/sbin` but is sometimes in `/usr/sbin`. Most recent Linux distributions start this daemon in the boot scripts, but it is worth making sure that it is running before you begin working with NFS (just type **ps aux | grep portmap**).

3.3.3. The Daemons

NFS serving is taken care of by five daemons: **rpc.nfsd**, which does most of the work; **rpc.lockd** and **rpc.statd**, which handle file locking; **rpc.mountd**, which handles the initial mount requests, and **rpc.rquotad**, which handles user file quotas on exported volumes. Starting with 2.2.18, **lockd** is called by **nfsd** upon demand, so you do not need to worry about starting it yourself. **statd** will need to be started separately. Most recent Linux distributions will have startup scripts for these daemons.

The daemons are all part of the nfs-utils package, and may be either in the `/sbin` directory or the `/usr/sbin` directory.

If your distribution does not include them in the startup scripts, then you should add them, configured to start in the following order:

rpc.portmap
rpc.mountd, **rpc.nfsd**
rpc.statd, **rpc.lockd** (if necessary), and **rpc.rquotad**

The nfs-utils package has sample startup scripts for RedHat and Debian. If you are using a different distribution, in general you can just copy the RedHat script, but you will probably have to take out the line that says:

```
    . ../init.d/functions
```

to avoid getting error messages.

3.4. Verifying that NFS is running

To do this, query the portmapper with the command **rpcinfo -p** to find out what services it is providing. You should get something like this:

```
    program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100011    1   udp    749  rquotad
    100011    2   udp    749  rquotad
    100005    1   udp    759  mountd
    100005    1   tcp    761  mountd
    100005    2   udp    764  mountd
    100005    2   tcp    766  mountd
    100005    3   udp    769  mountd
    100005    3   tcp    771  mountd
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    300019    1   tcp    830  amd
    300019    1   udp    831  amd
    100024    1   udp    944  status
    100024    1   tcp    946  status
    100021    1   udp   1042  nlockmgr
    100021    3   udp   1042  nlockmgr
    100021    4   udp   1042  nlockmgr
    100021    1   tcp   1629  nlockmgr
    100021    3   tcp   1629  nlockmgr
    100021    4   tcp   1629  nlockmgr
```

This says that we have NFS versions 2 and 3, rpc.statd version 1, network lock manager (the service name for rpc.lockd) versions 1, 3, and 4. There are also different service listings depending on whether NFS is travelling over TCP or UDP. Linux systems use UDP by default unless TCP is explicitly requested; however other OSes such as Solaris default to TCP.

If you do not at least see a line that says `portmapper`, a line that says `nfs`, and a line that says `mountd` then you will need to backtrack and try again to start up the daemons (see Section 7, Troubleshooting, if this still doesn't work).

If you do see these services listed, then you should be ready to set up NFS clients to access files from your server.

3.5. Making changes to /etc/exports later on

If you come back and change your `/etc/exports` file, the changes you make may not take effect immediately. You should run the command **exportfs -ra** to force **nfsd** to re-read the `/etc/exports` file. If you can't find the **exportfs** command, then you can kill **nfsd** with the **`-HUP`** flag (see the man pages for kill for details).

If that still doesn't work, don't forget to check `hosts.allow` to make sure you haven't forgotten to list any new client machines there. Also check the host listings on any firewalls you may have set up (see Section 7 and Section 6 for more details on firewalls and NFS).

4. Setting up an NFS Client

4.1. Mounting remote directories

Before beginning, you should double-check to make sure your mount program is new enough (version 2.10m if you want to use Version 3 NFS), and that the client machine supports NFS mounting, though most standard distributions do. If you are using a 2.2 or later kernel with the `/proc` filesystem you can check the latter by reading the file `/proc/filesystems` and making sure there is a line containing nfs. If not, typing **`insmod nfs`** may make it magically appear if NFS has been compiled as a module;

otherwise, you will need to build (or download) a kernel that has NFS support built in. In general, kernels that do not have NFS compiled in will give a very specific error when the **mount** command below is run.

To begin using machine as an NFS client, you will need the portmapper running on that machine, and to use NFS file locking, you will also need **rpc.statd** and **rpc.lockd** running on both the client and the server. Most recent distributions start those services by default at boot time; if yours doesn't, see Section 3.2 for information on how to start them up.

With **portmap**, **lockd**, and **statd** running, you should now be able to mount the remote directory from your server just the way you mount a local hard drive, with the mount command. Continuing our example from the previous section, suppose our server above is called *master.foo.com*,and we want to mount the `/home` directory on *slave1.foo.com*. Then, all we have to do, from the root prompt on *slave1.foo.com*, is type:

```
# mount master.foo.com:/home /mnt/home
```

and the directory `/home` on master will appear as the directory `/mnt/home` on *slave1*. (Note that this assumes we have created the directory `/mnt/home` as an empty mount point beforehand.)

If this does not work, see the Troubleshooting section (Section 7).

You can get rid of the file system by typing

```
# umount /mnt/home
```

just like you would for a local file system.

4.2. Getting NFS File Systems to Be Mounted at Boot Time

NFS file systems can be added to your `/etc/fstab` file the same way local file systems can, so that they mount when your system starts up. The only difference is that the file system type will be set to **`nfs`** and the dump and fsck order (the last two entries) will have to be set to zero. So for our example above, the entry in `/etc/fstab` would look like:

```
# device       mountpoint     fs-type     options      dump fsckorder
...
master.foo.com:/home  /mnt    nfs          rw            0    0
...
```

See the man pages for `fstab` if you are unfamiliar with the syntax of this file. If you are using an automounter such as amd or autofs, the options in the corresponding fields of your mount listings should look very similar if not identical.

At this point you should have NFS working, though a few tweaks may still be necessary to get it to work well. You should also read Section 6 to be sure your setup is reasonably secure.

4.3. Mount options

4.3.1. Soft vs. Hard Mounting

There are some options you should consider adding at once. They govern the way the NFS client handles a server crash or network outage. One of the cool things about NFS is that it can handle this gracefully. If you set up the clients right. There are two distinct failure modes:

soft

If a file request fails, the NFS client will report an error to the process on the client machine requesting the file access. Some programs can handle this with composure, most won't. We do not recommend using this setting; it is a recipe for corrupted files and lost data. You should especially not use this for mail disks --- if you value your mail, that is.

hard

The program accessing a file on a NFS mounted file system will hang when the server crashes. The process cannot be interrupted or killed (except by a "sure kill") unless you also specify **`intr`**. When the NFS server is back online the program will continue undisturbed from where it was. We recommend using **`hard,intr`** on all NFS mounted file systems.

Picking up the from previous example, the fstab entry would now look like:

```
# device             mountpoint   fs-type    options    dump fsckord
...
master.foo.com:/home  /mnt/home   nfs      rw,hard,intr  0     0
...
```

4.3.2. Setting Block Size to Optimize Transfer Speeds

The **`rsize`** and **`wsize`** mount options specify the size of the chunks of data that the client and server pass back and forth to each other.

The defaults may be too big or to small; there is no size that works well on all or most setups. On the one hand, some combinations of Linux kernels and network cards (largely on older machines) cannot handle blocks that large. On the other hand, if they can handle larger blocks, a bigger size might be faster.

Getting the block size right is an important factor in performance and is a must if you are planning to use the NFS server in a production environment. See Section 5 for details.

5. Optimizing NFS Performance

Careful analysis of your environment, both from the client and from the server point of view, is the first step necessary for optimal NFS performance. The first sections will address issues that are generally important to the client. Later (Section 5.3 and beyond), server side issues will be discussed. In both cases, these issues will not be limited exclusively to one side or the other, but it is useful to separate the two in order to get a clearer picture of cause and effect.

Aside from the general network configuration - appropriate network capacity, faster NICs, full duplex settings in order to reduce collisions, agreement in network speed among the switches and hubs, etc. - one of the most important client optimization settings are the NFS data transfer buffer sizes, specified by the **mount** command options **`rsize`** and **`wsize`**.

5.1. Setting Block Size to Optimize Transfer Speeds

The **mount** command options **`rsize`** and **`wsize`** specify the size of the chunks of data that the client and server pass back and forth to each other. If no **`rsize`** and **`wsize`** options are specified, the default varies by which version of NFS we are using. The most common default is 4K (4096 bytes), although for TCP-based mounts in 2.2 kernels, and for all mounts beginning with 2.4 kernels, the server specifies the default block size.

The theoretical limit for the NFS V2 protocol is 8K. For the V3 protocol, the limit is specific to the server. On the Linux server, the maximum block size is defined by the value of the kernel constant **`NFSSVC_MAXBLKSIZE`**, found in the Linux kernel source file `./include/linux/nfsd/const.h`. The current maximum block size for the kernel, as of 2.4.17, is 8K (8192 bytes), but the patch set implementing NFS over TCP/IP transport in the 2.4 series, as of this writing, uses a value of 32K (defined in the patch as 32*1024) for the maximum block size.

All 2.4 clients currently support up to 32K block transfer sizes, allowing the standard 32K block transfers across NFS mounts from other servers, such as Solaris, without client modification.

The defaults may be too big or too small, depending on the specific combination of hardware and kernels. On the one hand, some combinations of Linux kernels and network cards (largely on older machines) cannot handle blocks that large. On the other hand, if they can handle larger blocks, a bigger size might be faster.

You will want to experiment and find an **`rsize`** and **`wsize`** that works and is as fast as possible. You can test the speed of your options with some simple commands, if your network environment is not heavily used. Note that your results may vary widely unless you resort to using more complex benchmarks, such as Bonnie, Bonnie++, or IOzone.

The first of these commands transfers 16384 blocks of 16k each from the special file `/dev/zero` (which if you read it just spits out zeros *really* fast) to the mounted partition. We will time it to see how long it takes. So, from the client machine, type:

```
# time dd if=/dev/zero of=/mnt/home/testfile bs=16k count=16384
```

This creates a 256Mb file of zeroed bytes. In general, you should create a file that's at least twice as large as the system RAM on the server, but make sure you have enough disk space! Then read back the file into the great black hole on the client machine (`/dev/null`) by typing the following:

```
# time dd if=/mnt/home/testfile of=/dev/null bs=16k
```

Repeat this a few times and average how long it takes. Be sure to unmount and remount the filesystem each time (both on the client and, if you are zealous, locally on the server as well), which should clear out any caches.

Then unmount, and mount again with a larger and smaller block size. They should be multiples of 1024, and not larger than the maximum block size allowed by your system. Note that NFS Version 2 is limited to a maximum of 8K, regardless of the maximum block size defined by **`NFSSVC_MAXBLKSIZE`**; Version 3 will support up to 64K, if permitted. The block size should be a power of two since most of the parameters that would constrain it (such as file system block sizes and network packet size) are also powers of two. However, some users have reported better successes with block sizes that are not powers of two but are still multiples of the file system block size and the network packet size.

Directly after mounting with a larger size, cd into the mounted file system and do things like **ls**, explore the filesystem a bit to make sure everything is as it should. If the **`rsize`**/**`wsize`** is too large the symptoms are very odd and not 100% obvious. A typical symptom is incomplete file lists when doing **ls**, and no error messages, or reading files failing mysteriously with no error messages. After establishing that the given **`rsize`**/ **`wsize`** works you can do the speed tests again. Different server platforms are likely to have different optimal sizes.

Remember to edit `/etc/fstab` to reflect the **`rsize`**/**`wsize`** you found to be the most desirable.

If your results seem inconsistent, or doubtful, you may need to analyze your network more extensively while varying the **`rsize`** and **`wsize`** values. In that case, here are several pointers to benchmarks that may prove useful:

- Bonnie http://www.textuality.com/bonnie/
- Bonnie++ http://www.coker.com.au/bonnie++/
- IOzone file system benchmark http://www.iozone.org/
- The official NFS benchmark, SPECsfs97 http://www.spec.org/osg/sfs97/

The easiest benchmark with the widest coverage, including an extensive spread of file sizes, and of IO types - reads, & writes, rereads & rewrites, random access, etc. - seems to be IOzone. A recommended invocation of IOzone (for which you must have root privileges) includes unmounting and remounting the directory under test, in order to clear out the caches between tests, and including the file close time in the measurements. Assuming you've already exported `/tmp` to everyone from the server `foo`, and that you've installed IOzone in the local directory, this should work:

```
# echo "foo:/tmp /mnt/foo nfs rw,hard,intr,rsize=8192,wsize=8192 0 0"
>> /etc/fstab
# mkdir /mnt/foo
# mount /mnt/foo
# ./iozone -a -R -c -U /mnt/foo -f /mnt/foo/testfile > logfile
```

The benchmark should take 2-3 hours at most, but of course you will need to run it for each value of rsize and wsize that is of interest. The web site gives full documentation of the parameters, but the specific options used above are:

- **`-a`** Full automatic mode, which tests file sizes of 64K to 512M, using record sizes of 4K to 16M
- **`-R`** Generate report in excel spreadsheet form (The "surface plot" option for graphs is best)
- **`-c`** Include the file close time in the tests, which will pick up the NFS version 3 commit time
- **`-U`** Use the given mount point to unmount and remount between tests; it clears out caches
- **`-f`** When using unmount, you have to locate the test file in the mounted file system

5.2. Packet Size and Network Drivers

While many Linux network card drivers are excellent, some are quite shoddy, including a few drivers for some fairly standard cards. It is worth experimenting with your network card directly to find out how it can best handle traffic.

Try **ping**ing back and forth between the two machines with large packets using the **`-f`** and **`-s`** options with **ping** (see *ping(8)* for more details) and see if a lot of packets get dropped, or if they take a long time for a reply. If so, you may have a problem with the performance of your network card.

For a more extensive analysis of NFS behavior in particular, use the **nfsstat** command to look at nfs transactions, client and server statistics, network statistics, and so forth. The **`"-o net"`** option will show you the number of dropped packets in relation to the total number of transactions. In UDP transactions, the most important statistic is the number of retransmissions, due to dropped packets, socket buffer overflows, general server congestion, timeouts, etc. This will have a tremendously important effect on NFS performance, and should be carefully monitored. Note that **nfsstat** does not yet implement the **`-z`** option, which would zero out all counters, so you must look at the current **nfsstat** counter values prior to running the benchmarks.

To correct network problems, you may wish to reconfigure the packet size that your network card uses. Very often there is a constraint somewhere else in the network (such as a router) that causes a smaller maximum packet size between two machines than what the network cards on the machines are actually capable of. TCP should autodiscover the appropriate packet size for a network, but UDP will simply stay at a default value. So determining the appropriate packet size is especially important if you are using NFS over UDP.

You can test for the network packet size using the **tracepath** command: From the client machine, just type **`tracepath`** *server* **`2049`** and the path MTU should be reported at the bottom. You can then set the MTU on your network card equal to the path MTU, by using the **`MTU`** option to **ifconfig**, and see if fewer packets get dropped. See the **ifconfig** man pages for details on how to reset the MTU.

In addition, **netstat -s** will give the statistics collected for traffic across all supported protocols. You may also look at `/proc/net/snmp` for information about current network behavior; see the next section for more details.

5.3. Overflow of Fragmented Packets

Using an **`rsize`** or **`wsize`** larger than your network's MTU (often set to 1500, in many networks) will cause IP packet fragmentation when using NFS over UDP. IP packet fragmentation and reassembly require a significant amount of CPU resource at both ends of a network connection. In addition, packet fragmentation also exposes your network traffic to greater unreliability, since a complete RPC request must be retransmitted if a UDP packet fragment is dropped for any reason. Any increase of RPC retransmissions, along with the possibility of increased timeouts, are the single worst impediment to performance for NFS over UDP.

Packets may be dropped for many reasons. If your network topography is complex, fragment routes may differ, and may not all arrive at the Server for reassembly. NFS Server capacity may also be an issue, since the kernel has a limit of how many fragments it can buffer before it starts throwing away packets. With kernels that support the `/proc` filesystem, you can monitor the files

`/proc/sys/net/ipv4/ipfrag_high_thresh` and `/proc/sys/net/ipv4/ipfrag_low_thresh`. Once the number of unprocessed, fragmented packets reaches the number specified by `ipfrag_high_thresh` (in bytes), the kernel will simply start throwing away fragmented packets until the number of incomplete packets reaches the number specified by `ipfrag_low_thresh.`

Another counter to monitor is **`IP: ReasmFails`** in the file `/proc/net/snmp`; this is the number of fragment reassembly failures. if it goes up too quickly during heavy file activity, you may have problem.

5.4. NFS over TCP

A new feature, available for both 2.4 and 2.5 kernels but not yet integrated into the mainstream kernel at the time of this writing, is NFS over TCP. Using TCP has a distinct advantage and a distinct disadvantage over UDP. The advantage is that it works far better than UDP on lossy networks. When using TCP, a single dropped packet can be retransmitted, without the retransmission of the entire RPC request, resulting in better performance on lossy networks. In addition, TCP will handle network speed differences better than UDP, due to the underlying flow control at the network level.

The disadvantage of using TCP is that it is not a stateless protocol like UDP. If your server crashes in the middle of a packet transmission, the client will hang and any shares will need to be unmounted and remounted.

The overhead incurred by the TCP protocol will result in somewhat slower performance than UDP under ideal network conditions, but the cost is not severe, and is often not noticable without careful measurement. If you are using gigabit ethernet from end to end, you might also investigate the usage of jumbo frames, since the high speed network may allow the larger frame sizes without encountering increased collision rates, particularly if you have set the network to full duplex.

5.5. Timeout and Retransmission Values

Two mount command options, **`timeo`** and **`retrans`**, control the behavior of UDP requests when encountering client timeouts due to dropped packets, network congestion, and so forth. The **`-o timeo`** option allows designation of the length of time, in tenths of seconds, that the client will wait until it decides it will not get a reply from the server, and must try to send the request again. The default value is 7 tenths of a second. The **`-o retrans`** option allows designation of the number of timeouts allowed before the client gives up, and displays the `Server not responding` message. The default value is 3 attempts. Once the client displays this message, it will continue to try to send the request, but only once before displaying the error message if another timeout occurs. When the client reestablishes contact, it will fall back to using the correct **`retrans`** value, and will display the `Server OK` message.

If you are already encountering excessive retransmissions (see the output of the **nfsstat** command), or want to increase the block transfer size without encountering timeouts and retransmissions, you may want to adjust these values. The specific adjustment will depend upon your environment, and in most cases, the current defaults are appropriate.

5.6. Number of Instances of the NFSD Server Daemon

Most startup scripts, Linux and otherwise, start 8 instances of **nfsd**. In the early days of NFS, Sun decided on this number as a rule of thumb, and everyone else copied. There are no good measures of how many instances are optimal, but a more heavily-trafficked server may require more. You should use at the very least one daemon per processor, but four to eight per processor may be a better rule of thumb. If you are using a 2.4 or higher kernel and you want to see how heavily each **nfsd** thread is being used, you can look at the file `/proc/net/rpc/nfsd`. The last ten numbers on the **`th`** line in that file indicate the number of seconds that the thread usage was at that percentage of the maximum allowable. If you have a large number in the top three deciles, you may wish to increase the number of **nfsd** instances. This is done upon starting **nfsd** using the number of instances as the command line option, and is specified in the NFS startup script (`/etc/rc.d/init.d/nfs` on Red Hat) as **`RPCNFSDCOUNT`**. See the *nfsd(8)* man page for more information.

5.7. Memory Limits on the Input Queue

On 2.2 and 2.4 kernels, the socket input queue, where requests sit while they are currently being processed, has a small default size limit (`rmem_default`) of 64k. This queue is important for clients with heavy read loads, and servers with heavy write loads. As an example, if you are running 8 instances of nfsd on the server, each will only have 8k to store write requests while it processes them. In addition, the socket output queue - important for clients with heavy write loads and servers with heavy read loads - also has a small default size (`wmem_default`).

Several published runs of the NFS benchmark SPECsfs (http://www.spec.org/osg/sfs97/) specify usage of a much higher value for both the read and write value sets, `[rw]mem_default` and `[rw]mem_max`. You might consider increasing these values to at least 256k. The read and write limits are set in the proc file system using (for example) the files `/proc/sys/net/core/rmem_default` and `/proc/sys/net/core/rmem_max`. The `rmem_default` value can be increased in three steps; the following method is a bit of a hack but should work and should not cause any problems:

- Increase the size listed in the file:

  ```
  # echo 262144 > /proc/sys/net/core/rmem_default
  # echo 262144 > /proc/sys/net/core/rmem_max
  ```

- Restart NFS. For example, on Red Hat systems,

  ```
  # /etc/rc.d/init.d/nfs restart
  ```

- You might return the size limits to their normal size in case other kernel systems depend on it:

  ```
  # echo 65536 > /proc/sys/net/core/rmem_default
  # echo 65536 > /proc/sys/net/core/rmem_max
  ```

This last step may be necessary because machines have been reported to crash if these values are left changed for long periods of time.

5.8. Turning Off Autonegotiation of NICs and Hubs

If network cards auto-negotiate badly with hubs and switches, and ports run at different speeds, or with different duplex configurations, performance will be severely impacted due to excessive collisions, dropped packets, etc. If you see excessive numbers of dropped packets in the **nfsstat** output, or poor network performance in general, try playing around with the network speed and duplex settings. If possible, concentrate on establishing a 100BaseT full duplex subnet; the virtual elimination of collisions in full duplex will remove the most severe performance inhibitor for NFS over UDP. Be careful when turning off autonegotiation on a card: The hub or switch that the card is attached to will then resort to other mechanisms (such as parallel detection) to determine the duplex settings, and some cards default to half duplex because it is more likely to be supported by an old hub. The best solution, if the driver supports it, is to force the card to negotiate 100BaseT full duplex.

5.9. Synchronous vs. Asynchronous Behavior in NFS

The default export behavior for both NFS Version 2 and Version 3 protocols, used by **exportfs** in nfs-utils versions prior to Version 1.11 (the latter is in the CVS tree, but not yet released in a package, as of January, 2002) is "asynchronous". This default permits the server to reply to client requests as soon as it has processed the request and handed it off to the local file system, without waiting for the data to be written to stable storage. This is indicated by the **`async`** option denoted in the server's export list. It yields better performance at the cost of possible data corruption if the server reboots while still holding unwritten data and/or metadata in its caches. This possible data corruption is not detectable at the time of occurrence, since the **`async`** option instructs the server to lie to the client, telling the client that all data has indeed been written to the stable storage, regardless of the protocol used.

In order to conform with "synchronous" behavior, used as the default for most proprietary systems supporting NFS (Solaris, HP-UX, RS/6000, etc.), and now used as the default in the latest version of **exportfs**, the Linux Server's file system must be exported with the **`sync`** option. Note that specifying synchronous exports will result in no option being seen in the server's export list:

- Export a couple file systems to everyone, using slightly different options:

```
# /usr/sbin/exportfs -o rw,sync *:/usr/local
# /usr/sbin/exportfs -o rw *:/tmp
```

- Now we can see what the exported file system parameters look like:

```
# /usr/sbin/exportfs -v
/usr/local *(rw)
```

```
/tmp *(rw,async)
```

If your kernel is compiled with the `/proc` filesystem, then the file `/proc/fs/nfs/exports` will also show the full list of export options.

When synchronous behavior is specified, the server will not complete (that is, reply to the client) an NFS version 2 protocol request until the local file system has written all data/metadata to the disk. The server *will* complete a synchronous NFS version 3 request without this delay, and will return the status of the data in order to inform the client as to what data should be maintained in its caches, and what data is safe to discard. There are 3 possible status values, defined an enumerated type, **`nfs3_stable_how`**, in `include/linux/nfs.h`. The values, along with the subsequent actions taken due to these results, are as follows:

- NFS_UNSTABLE - Data/Metadata was not committed to stable storage on the server, and must be cached on the client until a subsequent client commit request assures that the server does send data to stable storage.
- NFS_DATA_SYNC - Metadata was not sent to stable storage, and must be cached on the client. A subsequent commit is necessary, as is required above.
- NFS_FILE_SYNC - No data/metadata need be cached, and a subsequent commit need not be sent for the range covered by this request.

In addition to the above definition of synchronous behavior, the client may explicitly insist on total synchronous behavior, regardless of the protocol, by opening all files with the **`O_SYNC`** option. In this case, all replies to client requests will wait until the data has hit the server's disk, regardless of the protocol used (meaning that, in NFS version 3, all requests will be **`NFS_FILE_SYNC`** requests, and will require that the Server returns this status). In that case, the performance of NFS Version 2 and NFS Version 3 will be virtually identical.

If, however, the old default **`async`** behavior is used, the **`O_SYNC`** option has no effect at all in either version of NFS, since the server will reply to the client without waiting for the write to complete. In that case the performance differences between versions will also disappear.

Finally, note that, for NFS version 3 protocol requests, a subsequent commit request from the NFS client at file close time, or at **fsync()** time, will force the server to write any previously unwritten data/metadata to the disk, and the server will not reply to the client until this has been completed, as long as **`sync`** behavior is followed. If **`async`** is used, the commit is essentially a no-op, since the server once again lies to the client, telling the client that the data has been sent to stable storage. This again exposes the client and server to data corruption, since cached data may be discarded on the client due to its belief that the server now has the data maintained in stable storage.

5.10. Non-NFS-Related Means of Enhancing Server Performance

In general, server performance and server disk access speed will have an important effect on NFS performance. Offering general guidelines for setting up a well-functioning file server is outside the scope of this document, but a few hints may be worth mentioning:

- If you have access to RAID arrays, use RAID 1/0 for both write speed and redundancy; RAID 5 gives you good read speeds but lousy write speeds.
- A journalling filesystem will drastically reduce your reboot time in the event of a system crash. Currently, ext3 (ftp://ftp.uk.linux.org/pub/linux/sct/fs/jfs/) will work correctly with NFS version 3. In addition, Reiserfs version 3.6 will work with NFS version 3 on 2.4.7 or later kernels (patches are available for previous kernels). Earlier versions of Reiserfs did not include room for generation numbers in the inode, exposing the possibility of undetected data corruption during a server reboot.
- Additionally, journalled file systems can be configured to maximize performance by taking advantage of the fact that journal updates are all that is necessary for data protection. One example is using ext3 with `data=journal` so that all updates go first to the journal, and later to the main file system. Once the journal has been updated, the NFS server can safely issue the reply to the clients, and the main file system update can occur at the server's leisure.

  The journal in a journalling file system may also reside on a separate device such as a flash memory card so that journal updates normally require no seeking. With only rotational delay imposing a cost, this gives reasonably good synchronous IO performance. Note that ext3 currently supports journal relocation, and ReiserFS will (officially) support it soon. The Reiserfs tool package found at ftp://ftp.namesys.com/pub/reiserfsprogs/reiserfsprogs-3.x.0k.tar.gz (ftp://ftp.namesys.com/pub/reiserfsprogs/reiserfsprogs-3.x.0k.tar.gz) contains the **reiserfstune** tool, which will allow journal relocation. It does, however, require a kernel patch which has not yet been officially released as of January, 2002.

- Using an automounter (such as autofs or amd) may prevent hangs if you cross-mount files on your machines (whether on purpose or by oversight) and one of those machines goes down. See the Automount Mini-HOWTO (http://www.linuxdoc.org/HOWTO/mini/Automount.html) for details.
- Some manufacturers (Network Appliance, Hewlett Packard, and others) provide NFS accelerators in the form of Non-Volatile RAM. NVRAM will boost access speed to stable storage up to the equivalent of `async` access.

6. Security and NFS

This list of security tips and explanations will not make your site completely secure. *NOTHING* will make your site completely secure. Reading this section may help you get an idea of the security problems with NFS. This is not a comprehensive guide and it will always be undergoing changes. If you have any tips or hints to give us please send them to the HOWTO maintainer.

If you are on a network with no access to the outside world (not even a modem) and you trust all the internal machines and all your users then this section will be of no use to you. However, its our belief that there are relatively few networks in this situation so we would suggest reading this section thoroughly for anyone setting up NFS.

With NFS, there are two steps required for a client to gain access to a file contained in a remote directory on the server. The first step is mount access. Mount access is achieved by the client machine attempting to attach to the server. The security for this is provided by the `/etc/exports` file. This file lists the names or IP addresses for machines that are allowed to access a share point. If the client's ip address matches one of the entries in the access list then it will be allowed to mount. This is not terribly secure. If someone is capable of spoofing or taking over a trusted address then they can access your mount points. To give a real-world example of this type of "authentication": This is equivalent to someone introducing themselves to you and you believing they are who they claim to be because they are wearing a sticker that says "Hello, My Name is ...." Once the machine has mounted a volume, its operating system will have access to all files on the volume (with the possible exception of those owned by root; see below) and write access to those files as well, if the volume was exported with the **`rw`** option.

The second step is file access. This is a function of normal file system access controls on the client and not a specialized function of NFS. Once the drive is mounted the user and group permissions on the files determine access control.

An example: bob on the server maps to the UserID 9999. Bob makes a file on the server that is only accessible the user (the equivalent to typing **`chmod 600`** *filename*). A client is allowed to mount the drive where the file is stored. On the client mary maps to UserID 9999. This means that the client user mary can access bob's file that is marked as only accessible by him. It gets worse: If someone has become superuser on the client machine they can **su -** *username* and become *any* user. NFS will be none the wiser.

Its not all terrible. There are a few measures you can take on the server to offset the danger of the clients. We will cover those shortly.

If you don't think the security measures apply to you, you're probably wrong. In Section 6.1 we'll cover securing the portmapper, server and client security in Section 6.2 and Section 6.3 respectively. Finally, in Section 6.4 we'll briefly talk about proper firewalling for your nfs server.

Finally, it is critical that all of your nfs daemons and client programs are current. If you think that a flaw is too recently announced for it to be a problem for you, then you've probably already been compromised.

A good way to keep up to date on security alerts is to subscribe to the bugtraq mailinglists. You can read up on how to subscribe and various other information about bugtraq here: http://www.securityfocus.com/forums/bugtraq/faq.html

Additionally searching for *NFS* at securityfocus.com's (http://www.securityfocus.com) search engine will show you all security reports pertaining to NFS.

You should also regularly check CERT advisories. See the CERT web page at www.cert.org (http://www.cert.org).

6.1. The portmapper

The portmapper keeps a list of what services are running on what ports. This list is used by a connecting machine to see what ports it wants to talk to access certain services.

The portmapper is not in as bad a shape as a few years ago but it is still a point of worry for many sys admins. The portmapper, like NFS and NIS, should not really have connections made to it outside of a trusted local area network. If you have to expose them to the outside world - be careful and keep up diligent monitoring of those systems.

Not all Linux distributions were created equal. Some seemingly up-to-date distributions do not include a securable portmapper. The easy way to check if your portmapper is good or not is to run *strings(1)* and see if it reads the relevant files, `/etc/hosts.deny` and `/etc/hosts.allow`. Assuming your portmapper is `/sbin/portmap` you can check it with this command:

```
strings /sbin/portmap | grep hosts.
```

On a securable machine it comes up something like this:

```
/etc/hosts.allow
/etc/hosts.deny
@(#) hosts_ctl.c 1.4 94/12/28 17:42:27
@(#) hosts_access.c 1.21 97/02/12 02:13:22
```

First we edit `/etc/hosts.deny`. It should contain the line

```
portmap: ALL
```

which will deny access to everyone. While it is closed run:

```
rpcinfo -p
```

just to check that your portmapper really reads and obeys this file. Rpcinfo should give no output, or possibly an error message. The files `/etc/hosts.allow` and `/etc/hosts.deny` take effect immediately after you save them. No daemon needs to be restarted.

Closing the portmapper for everyone is a bit drastic, so we open it again by editing `/etc/hosts.allow`. But first we need to figure out what to put in it. It should basically list all machines that should have access to your portmapper. On a run of the mill Linux system there are very few machines that need any access for any reason. The portmapper administers **nfsd**, **mountd**, **ypbind/ypserv**, **rquotad**, **lockd** (which shows up as `nlockmgr`), **statd** (which shows up as `status`) and 'r' services like **ruptime** and **rusers**. Of these only **nfsd**, **mountd**, **ypbind/ypserv** and perhaps **rquotad**,**lockd** and **statd** are of any consequence. All machines that need to access services on your machine should be allowed to do that. Let's say that your machine's address is *192.168.0.254* and that it lives on the subnet *192.168.0.0*, and that all machines on the subnet should have access to it (for an overview of those terms see the Networking-Overview-HOWTO (http://www.linuxdoc.org/HOWTO/Networking-Overview-HOWTO.html)). Then we write:

```
portmap: 192.168.0.0/255.255.255.0
```

in `/etc/hosts.allow`. If you are not sure what your network or netmask are, you can use the **ifconfig** command to determine the netmask and the **netstat** command to determine the network. For, example, for the device eth0 on the above machine **ifconfig** should show:

```
...
eth0   Link encap:Ethernet  HWaddr 00:60:8C:96:D5:56
       inet addr:192.168.0.254  Bcast:192.168.0.255 Mask:255.255.255.0
       UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
       RX packets:360315 errors:0 dropped:0 overruns:0
       TX packets:179274 errors:0 dropped:0 overruns:0
       Interrupt:10 Base address:0x320
...
```

and **netstat -rn** should show:

```
Kernel routing table
Destination     Gateway         Genmask         Flags Metric Ref Use    Iface
...
192.168.0.0     0.0.0.0         255.255.255.0   U     0      0   174412 eth0
...
```

(The network address is in the first column).

The `/etc/hosts.deny` and `/etc/hosts.allow` files are described in the manual pages of the same names.

IMPORTANT: Do not put anything but IP NUMBERS in the portmap lines of these files. Host name lookups can indirectly cause portmap activity which will trigger host name lookups which can indirectly cause portmap activity which will trigger...

Versions 0.2.0 and higher of the nfs-utils package also use the `hosts.allow` and `hosts.deny` files, so you should put in entries for **lockd**, **statd**, **mountd**, and **rquotad** in these files too. For a complete example, see Section 3.2.2.

The above things should make your server tighter. The only remaining problem is if someone gains administrative access to one of your trusted client machines and is able to send bogus NFS requests. The next section deals with safeguards against this problem.

6.2. Server security: nfsd and mountd

On the server we can decide that we don't want to trust any requests made as root on the client. We can do that by using the **`root_squash`** option in `/etc/exports`:

```
   /home slave1(rw,root_squash)
```

This is, in fact, the default. It should always be turned on unless you have a *very* good reason to turn it off. To turn it off use the **`no_root_squash`** option.

Now, if a user with *UID* 0 (i.e., root's user ID number) on the client attempts to access (read, write, delete) the file system, the server substitutes the *UID* of the server's 'nobody' account. Which means that the root user on the client can't access or change files that only root on the server can access or change. That's good, and you should probably use **`root_squash`** on all the file systems you export. "But the root user on the client can still use **su** to become any other user and access and change that users files!" say you. To which the answer is: Yes, and that's the way it is, and has to be with Unix and NFS. This has one important implication: All important binaries and files should be owned by root, and not bin or other non-root account, since the only account the clients root user cannot access is the servers root account. In the *exports(5)* man page there are several other squash options listed so that you can decide to mistrust whomever you (don't) like on the clients.

The TCP ports 1-1024 are reserved for root's use (and therefore sometimes referred to as "secure ports") A non-root user cannot bind these ports. Adding the **`secure`** option to an `/etc/exports` means that it will only listed to requests coming from ports 1-1024 on the client, so that a malicious non-root user on the client cannot come along and open up a spoofed NFS dialogue on a non-reserved port. This option is set by default.

6.3. Client Security

6.3.1. The nosuid mount option

On the client we can decide that we don't want to trust the server too much a couple of ways with options to mount. For example we can forbid suid programs to work off the NFS file system with the **`nosuid`** option. Some unix programs, such as passwd, are called "suid" programs: They set the id of the person running them to whomever is the owner of the file. If a file is owned by root and is suid, then the program will execute as root, so that they can perform operations (such as writing to the password file) that only root is allowed to do. Using the **`nosuid`** option is a good idea and you should consider using this with all NFS mounted disks. It means that the server's root user cannot make a suid-root program on the file system, log in to the client as a normal user and then use the suid-root program to become root on the client too. One could also forbid execution of files on the mounted file system altogether with the **`noexec`** option. But this is more likely to be impractical than **`nosuid`** since a file system is likely to at least contain some scripts or programs that need to be executed.

6.3.2. The broken_suid mount option

Some older programs (**xterm** being one of them) used to rely on the idea that root can write everywhere. This is will break under new kernels on NFS mounts. The security implications are that programs that do this type of suid action can potentially be used to change your apparent uid on nfs servers doing uid mapping. So the default has been to disable this **`broken_suid`** in the linux kernel.

The long and short of it is this: If you're using an old linux distribution, some sort of old suid program or an older unix of some type you *might* have to mount from your clients with the **`broken_suid`** option to **mount**. However, most recent unixes and linux distros have **xterm** and such programs just as a normal executable with no suid status, they call programs to do their setuid work.

You enter the above options in the options column, with the **`rsize`** and **`wsize`**, separated by commas.

6.3.3. Securing portmapper, rpc.statd, and rpc.lockd on the client

In the current (2.2.18+) implementation of NFS, full file locking is supported. This means that **rpc.statd** and **rpc.lockd** must be running on the client in order for locks to function correctly. These services require the portmapper to be running. So, most of the problems you will find with nfs on the server you may also be plagued with on the client. Read through the portmapper section above for information on securing the portmapper.

6.4. NFS and firewalls (ipchains and netfilter)

IPchains (under the 2.2.X kernels) and netfilter (under the 2.4.x kernels) allow a good level of security - instead of relying on the daemon (or perhaps its TCP wrapper) to determine which machines can connect, the connection attempt is allowed or disallowed at a lower level. In this case, you can stop the connection much earlier and more globally, which can protect you from all sorts of attacks.

Describing how to set up a Linux firewall is well beyond the scope of this document. Interested readers may wish to read the Firewall-HOWTO (http://www.linuxdoc.org/HOWTO/Firewall-HOWTO.html) or the IPCHAINS-HOWTO (http://www.linuxdoc.org/HOWTO/IPCHAINS-HOWTO.HTML). For users of kernel 2.4 and above you might want to visit the netfilter webpage at: http://netfilter.filewatcher.org. If you are already familiar with the workings of ipchains or netfilter this section will give you a few tips on how to better setup your NFS daemons to more easily firewall and protect them.

A good rule to follow for your firewall configuration is to deny all, and allow only some - this helps to keep you from accidentally allowing more than you intended.

In order to understand how to firewall the NFS daemons, it will help to breifly review how they bind to ports.

When a daemon starts up, it requests a free port from the portmapper. The portmapper gets the port for the daemon and keeps track of the port currently used by that daemon. When other hosts or processes need to communicate with the daemon, they request the port number from the portmapper in order to find the daemon. So the ports will perpetually float because different ports may be free at different times and so the portmapper will allocate them differently each time. This is a pain for setting up a firewall. If you never know where the daemons are going to be then you don't know precisely which ports to allow access to. This might not be a big deal for many people running on a protected or isolated LAN. For those people on a public network, though, this is horrible.

In kernels 2.4.13 and later with nfs-utils 0.3.3 or later you no longer have to worry about the floating of ports in the portmapper. Now all of the daemons pertaining to nfs can be "pinned" to a port. Most of them nicely take a **-p** option when they are started; those daemons that are started by the kernel take some kernel arguments or module options. They are described below.

Some of the daemons involved in sharing data via nfs are already bound to a port. **portmap** is always on port 111 tcp and udp. **nfsd** is always on port 2049 TCP and UDP (however, as of kernel 2.4.17, NFS over TCP is considered experimental and is not for use on production machines).

The other daemons, **statd**, **mountd**, **lockd**, and **rquotad**, will normally move around to the first available port they are informed of by the portmapper.

To force **statd** to bind to a particular port, use the `-p` *portnum* option. To force **statd** to respond on a particular port, additionally use the `-o` *portnum* option when starting it.

To force **mountd** to bind to a particular port use the **-p** *portnum* option.

For example, to have statd broadcast of port 32765 and listen on port 32766, and mountd listen on port 32767, you would type:

```
# statd -p 32765 -o 32766
# mountd -p 32767
```

lockd is started by the kernel when it is needed. Therefore you need to pass module options (if you have it built as a module) or kernel options to force **lockd** to listen and respond only on certain ports.

If you are using loadable modules and you would like to specify these options in your `/etc/modules.conf` file add a line like this to the file:

```
options lockd nlm_udpport=32768 nlm_tcpport=32768
```

The above line would specify the udp and tcp port for **lockd** to be 32768.

If you are not using loadable modules or if you have compiled **lockd** into the kernel instead of building it as a module then you will need to pass it an option on the kernel boot line.

It should look something like this:

```
 vmlinuz 3 root=/dev/hda1 lockd.udpport=32768 lockd.tcpport=32768
```

The port numbers do not have to match but it would simply add unnecessary confusion if they didn't.

If you are using quotas and using **rpc.quotad** to make these quotas viewable over nfs you will need to also take it into account when setting up your firewall. There are two **rpc.rquotad** source trees. One of those is maintained in the nfs-utils tree. The other in the quota-tools tree. They do not operate identically. The one provided with nfs-utils supports binding the daemon to a port with the **-p** directive. The one in quota-tools does not. Consult your distribution's documentation to determine if yours does.

For the sake of this discussion lets describe a network and setup a firewall to protect our nfs server. Our nfs server is 192.168.0.42 our client is 192.168.0.45 only. As in the example above, **statd** has been started so that it only binds to port 32765 for incoming requests and it must answer on port 32766. **mountd** is forced to bind to port 32767. **lockd**'s module parameters have been set to bind to 32768. **nfsd** is, of course, on port 2049 and the portmapper is on port 111.

We are not using quotas.

Using IPCHAINS, a simple firewall might look something like this:

```
ipchains -A input -f -j ACCEPT -s 192.168.0.45
ipchains -A input -s 192.168.0.45 -d 0/0 32765:32768 -p 6 -j ACCEPT
ipchains -A input -s 192.168.0.45 -d 0/0 32765:32768 -p 17 -j ACCEPT
ipchains -A input -s 192.168.0.45 -d 0/0 2049 -p 17 -j ACCEPT
ipchains -A input -s 192.168.0.45 -d 0/0 2049 -p 6 -j ACCEPT
ipchains -A input -s 192.168.0.45 -d 0/0 111 -p 6 -j ACCEPT
ipchains -A input -s 192.168.0.45 -d 0/0 111 -p 17 -j ACCEPT
ipchains -A input -s 0/0 -d 0/0 -p 6 -j DENY -y -l
ipchains -A input -s 0/0 -d 0/0 -p 17 -j DENY -l
```

The equivalent set of commands in netfilter is:

```
iptables -A INPUT -f -j ACCEPT -s 192.168.0.45
iptables -A INPUT -s 192.168.0.45 -d 0/0 32765:32768 -p 6 -j ACCEPT
iptables -A INPUT -s 192.168.0.45 -d 0/0 32765:32768 -p 17 -j ACCEPT
iptables -A INPUT -s 192.168.0.45 -d 0/0 2049 -p 17 -j ACCEPT
iptables -A INPUT -s 192.168.0.45 -d 0/0 2049 -p 6 -j ACCEPT
iptables -A INPUT -s 192.168.0.45 -d 0/0 111 -p 6 -j ACCEPT
iptables -A INPUT -s 192.168.0.45 -d 0/0 111 -p 17 -j ACCEPT
iptables -A INPUT -s 0/0 -d 0/0 -p 6 -j DENY --syn --log-level 5
iptables -A INPUT -s 0/0 -d 0/0 -p 17 -j DENY --log-level 5
```

The first line says to accept all packet fragments (except the first packet fragment which will be treated as a normal packet). In theory no packet will pass through until it is reassembled, and it won't be reassembled unless the first packet fragment is passed. Of course there are attacks that can be generated by overloading a machine with packet fragments. But NFS won't work correctly unless you let fragments through. See Section 7.8 for details.

The other lines allow specific connections from any port on our client host to the specific ports we have made available on our server. This means that if, say, 192.158.0.46 attempts to contact the NFS server it will not be able to mount or see what mounts are available.

With the new port pinning capabilities it is obviously much easier to control what hosts are allowed to mount your NFS shares. It is worth mentioning that NFS is not an encrypted protocol and anyone on the same physical network could sniff the traffic and reassemble the information being passed back and forth.

6.5. Tunneling NFS through SSH

One method of encrypting NFS traffic over a network is to use the port-forwarding capabilities of **ssh**. However, as we shall see, doing so has a serious drawback if you do not utterly and completely trust the local users on your server.

The first step will be to export files to the localhost. For example, to export the `/home` partition, enter the following into `/etc/exports`:

```
/home   127.0.0.1(rw)
```

The next step is to use **ssh** to forward ports. For example, **ssh** can tell the server to forward to any port on any machine from a port on the client. Let us assume, as in the previous section, that our server is 192.168.0.42, and that we have pinned **mountd** to port 32767 using the argument **`-p 32767`**. Then, on the client, we'll type:

```
# ssh root@192.168.0.42 -L 250:localhost:2049  -f sleep 60m
# ssh root@192.168.0.42 -L 251:localhost:32767 -f sleep 60m
```

The above command causes **ssh** on the client to take any request directed at the client's port 250 and forward it, first through **sshd** on the server, and then on to the server's port 2049. The second line causes a similar type of forwarding between requests to port 251 on the client and port 32767 on the server. The **`localhost`** is relative to the server; that is, the forwarding will be done to the server itself. The port could otherwise have been made to forward to any other machine, and the requests would look to the outside world as if they were coming from the server. Thus, the requests will appear to NFSD on the server as if they are coming from the server itself. Note that in order to bind to a port below 1024 on the client, we have to run this command as root on the client. Doing this will be necessary if we have exported our filesystem with the default **`secure`** option.

Finally, we are pulling a little trick with the last option, **`-f sleep 60m`**. Normally, when we use **ssh**, even with the **`-L`** option, we will open up a shell on the remote machine. But instead, we just want the port forwarding to execute in the background so that we get our shell on the client back. So, we tell **ssh** to execute a command in the background on the server to sleep for 60 minutes. This will cause the port to be forwarded for 60 minutes until it gets a connection; at that point, the port will continue to be forwarded until the connection dies or until the 60 minutes are up, whichever happens later. The above command could be put in our startup scripts on the client, right after the network is started.

Next, we have to mount the filesystem on the client. To do this, we tell the client to mount a filesystem on the localhost, but at a different port from the usual 2049. Specifically, an entry in `/etc/fstab` would look like:

```
  localhost:/home  /mnt/home  nfs  rw,hard,intr,port=250,mountport=251  0 0
```

Having done this, we can see why the above will be incredibly insecure if we have *any* ordinary users who are able to log in to the server locally. If they can, there is nothing preventing them from doing what we did and using **ssh** to forward a privileged port on their own client machine (where they are legitimately root) to ports 2049 and 32767 on the server. Thus, any ordinary user on the server can mount our filesystems with the same rights as root on our client.

If you are using an NFS server that does not have a way for ordinary users to log in, and you wish to use this method, there are two additional caveats: First, the connection travels from the client to the server via **sshd**; therefore you will have to leave port 22 (where **sshd** listens) open to your client on the firewall. However you do not need to leave the other ports, such as 2049 and 32767, open anymore. Second, file locking will no longer work. It is not possible to ask **statd** or the locking manager to make requests to a particular port for a particular mount; therefore, any locking requests will cause **statd** to connect to **statd** on localhost, i.e., itself, and it will fail with an error. Any attempt to correct this would require a major rewrite of NFS.

It may also be possible to use IPSec to encrypt network traffic between your client and your server, without compromising any local security on the server; this will not be taken up here. See the FreeS/WAN (http://www.freeswan.org/) home page for details on using IPSec under Linux.

6.6. Summary

If you use the `hosts.allow`, `hosts.deny`, **`root_squash`**, **`nosuid`** and privileged port features in the portmapper/NFS software, you avoid many of the presently known bugs in NFS and can almost feel secure about that at least. But still, after all that: When an intruder has access to your network, s/he can make strange commands appear in your `.forward` or read your mail when `/home` or `/var/mail` is NFS exported. For the same reason, you should never access your PGP private key over NFS. Or at least you should know the risk involved. And now you know a bit of it.

NFS and the portmapper makes up a complex subsystem and therefore it's not totally unlikely that new bugs will be discovered, either in the basic design or the implementation we use. There might even be holes known now, which someone is abusing. But that's life.

7. Troubleshooting

This is intended as a step-by-step guide to what to do when things go wrong using NFS. Usually trouble first rears its head on the client end, so this diagnostic will begin there.

7.1. Unable to See Files on a Mounted File System

First, check to see if the file system is actually mounted. There are several ways of doing this. The most reliable way is to look at the file `/proc/mounts`, which will list all mounted filesystems and give details about them. If this doesn't work (for example if you don't have the `/proc` filesystem compiled into your kernel), you can type **`mount -f`** although you get less information.

If the file system appears to be mounted, then you may have mounted another file system on top of it (in which case you should unmount and remount both volumes), or you may have exported the file system on the server before you mounted it there, in which case NFS is exporting the underlying mount point (if so then you need to restart NFS on the server).

If the file system is not mounted, then attempt to mount it. If this does not work, see *Symptom 3*.

7.2. File requests hang or timeout waiting for access to the file.

This usually means that the client is unable to communicate with the server. See *Symptom 3* letter b.

7.3. Unable to mount a file system

There are two common errors that mount produces when it is unable to mount a volume. These are:

a. failed, reason given by server: `Permission denied`

   This means that the server does not recognize that you have access to the volume.

   i. Check your `/etc/exports` file and make sure that the volume is exported and that your client has the right kind of access to it. For example, if a client only has read access then you have to mount the volume with the **`ro`** option rather than the **`rw`** option.
   ii. Make sure that you have told NFS to register any changes you made to `/etc/exports` since starting nfsd by running the exportfs command. Be sure to type **exportfs -ra** to be extra certain that the exports are being re-read.
   iii. Check the file `/proc/fs/nfs/exports` and make sure the volume and client are listed correctly. (You can also look at the file `/var/lib/nfs/xtab` for an unabridged list of how all the active export options are set.) If they are not, then you have not re-exported properly. If they are listed, make sure the server recognizes your client as being the machine you think it is. For example, you may have an old listing for the client in `/etc/hosts` that is throwing off the server, or you may not have listed the client's complete address and it may be resolving to a machine in a different domain. One trick is login to the server from the client via **ssh** or **telnet**; if you then type **who**, one of the listings should be your login session and the name of your client machine as the server sees it. Try using this machine name in your `/etc/exports` entry. Finally, try to ping the client from the server, and try to **ping** the server from the client. If this doesn't work, or if there is packet loss, you may have lower-level network problems.
   iv. It is not possible to export both a directory and its child (for example both `/usr` and `/usr/local`). You should export the parent directory with the necessary permissions, and all of its subdirectories can then be mounted with those same permissions.

b. `RPC: Program Not Registered`: (or another "RPC" error):

This means that the client does not detect NFS running on the server. This could be for several reasons.

i. First, check that NFS actually is running on the server by typing **rpcinfo -p** on the server. You should see something like this:

   ```
   program vers proto   port
    100000    2   tcp    111  portmapper
    100000    2   udp    111  portmapper
    100011    1   udp    749  rquotad
    100011    2   udp    749  rquotad
    100005    1   udp    759  mountd
    100005    1   tcp    761  mountd
    100005    2   udp    764  mountd
    100005    2   tcp    766  mountd
    100005    3   udp    769  mountd
    100005    3   tcp    771  mountd
    100003    2   udp   2049  nfs
    100003    3   udp   2049  nfs
    300019    1   tcp    830  amd
    300019    1   udp    831  amd
    100024    1   udp    944  status
    100024    1   tcp    946  status
    100021    1   udp   1042  nlockmgr
    100021    3   udp   1042  nlockmgr
    100021    4   udp   1042  nlockmgr
    100021    1   tcp   1629  nlockmgr
    100021    3   tcp   1629  nlockmgr
    100021    4   tcp   1629  nlockmgr
   ```

   This says that we have NFS versions 2 and 3, rpc.statd version 1, network lock manager (the service name for **rpc.lockd**) versions 1, 3, and 4. There are also different service listings depending on whether NFS is travelling over TCP or UDP. UDP is usually (but not always) the default unless TCP is explicitly requested.

   If you do not see at least `portmapper`, `nfs`, and `mountd`, then you need to restart NFS. If you are not able to restart successfully, proceed to *Symptom 9*.

ii. Now check to make sure you can see it from the client. On the client, type **rpcinfo -p** *server* where *server* is the DNS name or IP address of your server.

   If you get a listing, then make sure that the type of mount you are trying to perform is supported. For example, if you are trying to mount using Version 3 NFS, make sure Version 3 is listed; if you are trying to mount using NFS over TCP, make sure that is registered. (Some non-Linux clients default to TCP). Type **`man rpcinfo`** for more details on how to read the output. If the type of mount you are trying to perform is not listed, try a different type of mount.

   If you get the error `No Remote Programs Registered`, then you need to check your `/etc/hosts.allow` and `/etc/hosts.deny` files on the server and make sure your client actually is allowed access. Again, if the entries appear correct, check `/etc/hosts` (or your DNS server) and make sure that the machine is listed correctly, and make sure you can ping the server from the client. Also check the error logs on the system for helpful messages: Authentication errors from bad `/etc/hosts.allow` entries will usually appear in `/var/log/messages`, but may appear somewhere else depending on how your system logs are set up. The man pages for `syslog` can help you figure out how your logs are set up. Finally, some older operating systems may behave badly when routes between the two machines are asymmetric. Try typing **tracepath [server]** from the client and see if the word "asymmetric" shows up anywhere in the output. If it does then this may be causing packet loss. However asymmetric routes are not usually a problem on recent linux distributions.

If you get the error `Remote system error - No route to host`, but you can ping the server correctly, then you are the victim of an overzealous firewall. Check any firewalls that may be set up, either on the server or on any routers in between the client and the server. Look at the man pages for **ipchains**, **netfilter**, and **ipfwadm**, as well as the IPChains-HOWTO (http://www.linuxdoc.org/HOWTO/IPCHAINS-HOWTO.html) and the Firewall-HOWTO (http://www.linuxdoc.org/HOWTO/Firewall-HOWTO.html) for help.

7.4. I do not have permission to access files on the mounted volume.

This could be one of two problems.

If it is a write permission problem, check the export options on the server by looking at `/proc/fs/nfs/exports` and make sure the filesystem is not exported read-only. If it is you will need to re-export it read/write (don't forget to run **exportfs -ra** after editing `/etc/exports`). Also, check `/proc/mounts` and make sure the volume is mounted read/write (although if it is mounted read-only you ought to get a more specific error message). If not then you need to re-mount with the **`rw`** option.

The second problem has to do with username mappings, and is different depending on whether you are trying to do this as root or as a non-root user.

If you are not root, then usernames may not be in sync on the client and the server. Type **id [user]** on both the client and the server and make sure they give the same *UID* number. If they don't then you are having problems with NIS, NIS+, rsync, or whatever system you use to sync usernames. Check group names to make sure that they match as well. Also, make sure you are not exporting with the **`all_squash`** option. If the user names match then the user has a more general permissions problem unrelated to NFS.

If you are root, then you are probably not exporting with the **`no_root_squash`** option; check `/proc/fs/nfs/exports` or `/var/lib/nfs/xtab` on the server and make sure the option is listed. In general, being able to write to the NFS server as root is a bad idea unless you have an urgent need -- which is why Linux NFS prevents it by default. See Section 6 for details.

If you have root squashing, you want to keep it, and you're only trying to get root to have the same permissions on the file that the user *nobody* should have, then remember that it is the server that determines which uid root gets mapped to. By default, the server uses the *UID* and *GID* of *nobody* in the `/etc/passwd` file, but this can also be overridden with the **`anonuid`** and **`anongid`** options in the `/etc/exports` file. Make sure that the client and the server agree about which *UID nobody* gets mapped to.

7.5. When I transfer really big files, NFS takes over all the CPU cycles on the server and it screeches to a halt.

This is a problem with the `fsync()` function in 2.2 kernels that causes all sync-to-disk requests to be cumulative, resulting in a write time that is quadratic in the file size. If you can, upgrading to a 2.4 kernel should solve the problem. Also, exporting with the **`no_wdelay`** option forces the program to use `o_sync()` instead, which may prove faster.

7.6. Strange error or log messages

a. Messages of the following format:

```
Jan 7 09:15:29 server kernel: fh_verify: mail/guest permission failure, acc=4, error=13
Jan 7 09:23:51 server kernel: fh_verify: ekonomi/test permission failure, acc=4, error=
```

   These happen when a NFS `setattr` operation is attempted on a file you don't have write access to. The messages are harmless.

b. The following messages frequently appear in the logs:

```
kernel: nfs: server server.domain.name not responding, still trying
kernel: nfs: task 10754 can't get a request slot
kernel: nfs: server server.domain.name OK
```

   The "can't get a request slot" message means that the client-side RPC code has detected a lot of timeouts (perhaps due to network congestion, perhaps due to an overloaded server), and is throttling back the number of concurrent outstanding requests in an attempt to lighten the load. The cause of these messages is basically sluggish performance. See Section 5 for details.

c. After mounting, the following message appears on the client:

   ```
   nfs warning: mount version older than kernel
   ```

   It means what it says: You should upgrade your mount package and/or am-utils. (If for some reason upgrading is a problem, you may be able to get away with just recompiling them so that the newer kernel features are recognized at compile time).

d. Errors in startup/shutdown log for **lockd**

   You may see a message of the following kind in your boot log:

   ```
   nfslock: rpc.lockd startup failed
   ```

   They are harmless. Older versions of **rpc.lockd** needed to be started up manually, but newer versions are started automatically by **nfsd**. Many of the default startup scripts still try to start up **lockd** by hand, in case it is necessary. You can alter your startup scripts if you want the messages to go away.

e. The following message appears in the logs:

   ```
   kmem_create: forcing size word alignment - nfs_fh
   ```

   This results from the file handle being 16 bits instead of a mulitple of 32 bits, which makes the kernel grimace. It is harmless.

7.7. Real permissions don't match what's in `/etc/exports`.

`/etc/exports` is *very* sensitive to whitespace - so the following statements are not the same:

```
/export/dir hostname(rw,no_root_squash)
/export/dir hostname (rw,no_root_squash)
```

The first will grant **`hostname rw`** access to `/export/dir` without squashing root privileges. The second will grant **`hostname rw`** privileges with **`root squash`** and it will grant *everyone* else read/write access, without squashing root privileges. Nice huh?

7.8. Flaky and unreliable behavior

Simple commands such as **ls** work, but anything that transfers a large amount of information causes the mount point to lock.

This could be one of two problems:

i. It will happen if you have ipchains on at the server and/or the client and you are not allowing fragmented packets through the chains. Allow fragments from the remote host and you'll be able to function again. See Section 6.4 for details on how to do this.
ii. You may be using a larger **`rsize`** and **`wsize`** in your mount options than the server supports. Try reducing **`rsize`** and **`wsize`** to 1024 and seeing if the problem goes away. If it does, then increase them slowly to a more reasonable value.

7.9. nfsd won't start

Check the file `/etc/exports` and make sure root has read permission. Check the binaries and make sure they are executable. Make sure your kernel was compiled with NFS server support. You may need to reinstall your binaries if none of these ideas helps.

7.10. File Corruption When Using Multiple Clients

If a file has been modified within one second of its previous modification and left the same size, it will continue to generate the same inode number. Because of this, constant reads and writes to a file by multiple clients may cause file corruption. Fixing this bug requires changes deep within the filesystem layer, and therefore it is a 2.5 item.

8. Using Linux NFS with Other OSes

Every operating system, Linux included, has quirks and deviations in the behavior of its NFS implementation -- sometimes because the protocols are vague, sometimes because they leave gaping security holes. Linux will work properly with all major vendors' NFS implementations, as far as we know. However, there may be extra steps involved to make sure the two OSes are communicating clearly with one another. This section details those steps.

In general, it is highly ill-advised to attempt to use a Linux machine with a kernel before 2.2.18 as an NFS server for non-Linux clients. Implementations with older kernels may work fine as clients; however if you are using one of these kernels and get stuck, the first piece of advice we would give is to upgrade your kernel and see if the problems go away. The user-space NFS implementations also do not work well with non-Linux clients.

Following is a list of known issues for using Linux together with major operating systems.

8.1. AIX

8.1.1. Linux Clients and AIX Servers

The format for the `/etc/exports` file for our example in Section 3 is:

```
/usr   slave1.foo.com:slave2.foo.com,access=slave1.foo.com:slave2.foo.com
/home  slave1.foo.com:slave2.foo.com,rw=slave1.foo.com:slave2.foo.com
```

8.1.2. AIX clients and Linux Servers

AIX uses the file `/etc/filesystems` instead of `/etc/fstab`. A sample entry, based on the example in Section 4, looks like this:

```
/mnt/home:
        dev             = "/home"
        vfs             = nfs
        nodename        = master.foo.com
        mount           = true
        options         = bg,hard,intr,rsize=1024,wsize=1024,vers=2,proto=udp
        account         = false
```

i. Version 4.3.2 of AIX, and possibly earlier versions as well, requires that file systems be exported with the **`insecure`** option, which causes NFS to listen to requests from insecure ports (i.e., ports above 1024, to which non-root users can bind). Older versions of AIX do not seem to require this.

ii. AIX clients will default to mounting version 3 NFS over TCP. If your Linux server does not support this, then you may need to specify **`vers=2`** and/or **`proto=udp`** in your mount options.

iii. Using netmasks in `/etc/exports` seems to sometimes cause clients to lose mounts when another client is reset. This can be fixed by listing out hosts explicitly.

iv. Apparently automount in AIX 4.3.2 is rather broken.

8.2. BSD

8.2.1. BSD servers and Linux clients

BSD kernels tend to work better with larger block sizes.

8.2.2. Linux servers and BSD clients

Some versions of BSD may make requests to the server from insecure ports, in which case you will need to export your volumes with the **`insecure`** option. See the man page for *exports(5)* for more details.

8.3. Tru64 Unix

8.3.1. Tru64 Unix Servers and Linux Clients

In general, Tru64 Unix servers work quite smoothly with Linux clients. The format for the `/etc/exports` file for our example in Section 3 is:

```
/usr         slave1.foo.com:slave2.foo.com \
     -access=slave1.foo.com:slave2.foo.com \

/home        slave1.foo.com:slave2.foo.com \
         -rw=slave1.foo.com:slave2.foo.com \
       -root=slave1.foo.com:slave2.foo.com
```

(The **`root`** option is listed in the last entry for informational purposes only; its use is not recommended unless necessary.)

Tru64 checks the `/etc/exports` file every time there is a mount request so you do not need to run the **exportfs** command; in fact on many versions of Tru64 Unix the command does not exist.

8.3.2. Linux Servers and Tru64 Unix Clients

There are two issues to watch out for here. First, Tru64 Unix mounts using Version 3 NFS by default. You will see mount errors if your Linux server does not support Version 3 NFS. Second, in Tru64 Unix 4.x, NFS locking requests are made by `daemon`. You will therefore need to specify the **`insecure_locks`** option on all volumes you export to a Tru64 Unix 4.x client; see the **exports** man pages for details.

8.4. HP-UX

8.4.1. HP-UX Servers and Linux Clients

A sample `/etc/exports` entry on HP-UX looks like this:

```
/usr -ro,access=slave1.foo.com:slave2.foo.com
/home -rw=slave1.foo.com:slave2.fo.com:root=slave1.foo.com:slave2.foo.com
```

(The **`root`** option is listed in the last entry for informational purposes only; its use is not recommended unless necessary.)

8.4.2. Linux Servers and HP-UX Clients

HP-UX diskless clients will require at least a kernel version 2.2.19 (or patched 2.2.18) for device files to export correctly. Also, any exports to an HP-UX client will need to be exported with the **`insecure_locks`** option.

8.5. IRIX

8.5.1. IRIX Servers and Linux Clients

A sample `/etc/exports` entry on IRIX looks like this:

```
/usr -ro,access=slave1.foo.com:slave2.foo.com
/home -rw=slave1.foo.com:slave2.fo.com:root=slave1.foo.com:slave2.foo.com
```

(The **`root`** option is listed in the last entry for informational purposes only; its use is not recommended unless necessary.)

There are reportedly problems when using the nohide option on exports to linux 2.2-based systems. This problem is fixed in the 2.4 kernel. As a workaround, you can export and mount lower-down file systems separately.

As of Kernel 2.4.17, there continue to be several minor interoperability issues that may require a kernel upgrade. In particular:

- Make sure that Trond Myklebust's seekdir (or dir) kernel patch is applied. The latest version (for 2.4.17) is located at:

  http://www.fys.uio.no/~trondmy/src/2.4.17/linux-2.4.17-seekdir.dif (http://www.fys.uio.no/~trondmy/src/2.4.17/linux-2.4.17-seekdir.dif)

- IRIX servers do not always use the same `fsid` attribute field across reboots, which results in `inode number mismatch` errors on a Linux client if the mounted IRIX server reboots. A patch is available from:

  http://www.geocrawler.com/lists/3/SourceForge/789/0/7777454/ (http://www.geocrawler.com/lists/3/SourceForge/789/0/7777454/)

- Linux kernels v2.4.9 and above have problems reading large directories (hundreds of files) from exported IRIX XFS file systems that were made with **`naming version=1`**. The reason for the problem can be found at:

  http://www.geocrawler.com/archives/3/789/2001/9/100/6531172/ (http://www.geocrawler.com/archives/3/789/2001/9/100/6531172/)

  The naming version can be found by using (on the IRIX server):

  ```
  xfs_growfs -n mount_point
  ```

  The workaround is to export these file systems using the **`-32bitclients`** option in the `/etc/exports` file. The fix is to convert the file system to 'naming version=2'. Unfortunately the only way to do this is by a **`backup/mkfs/restore`**.

  mkfs_xfs on IRIX 6.5.14 (and above) creates **`naming version=2`** XFS file systems by default. On IRIX 6.5.5 to 6.5.13, use:

  ```
  mkfs_xfs -n version=2 device
  ```

  Versions of IRIX prior to 6.5.5 do not support **`naming version=2`** XFS file systems.

8.5.2. IRIX clients and Linux servers

Irix versions up to 6.5.12 have problems mounting file systems exported from Linux boxes - the mount point "gets lost," e.g.,

```
# mount linux:/disk1 /mnt
# cd /mnt/xyz/abc
# pwd
/xyz/abc
```

This is known IRIX bug (SGI bug 815265 - IRIX not liking file handles of less than 32 bytes), which is fixed in IRIX 6.5.13. If it is not possible to upgrade to IRIX 6.5.13, then the unofficial workaround is to force the Linux **nfsd** to always use 32 byte file handles.

A number of patches exist - see:

- http://www.geocrawler.com/archives/3/789/2001/8/50/6371896/ (http://www.geocrawler.com/archives/3/789/2001/8/50/6371896/)
- http://oss.sgi.com/projects/xfs/mail_archive/0110/msg00006.html (http://oss.sgi.com/projects/xfs/mail_archive/0110/msg00006.html)

8.6. Solaris

8.6.1. Solaris Servers

Solaris has a slightly different format on the server end from other operating systems. Instead of `/etc/exports`, the configuration file is `/etc/dfs/dfstab`. Entries are of the form of a **share** command, where the syntax for the example in Section 3 would look like

```
share -o rw=slave1,slave2 -d "Master Usr" /usr
```

and instead of running **exportfs** after editing, you run **shareall**.

Solaris servers are especially sensitive to packet size. If you are using a Linux client with a Solaris server, be sure to set **`rsize`** and **`wsize`** to 32768 at mount time.

Finally, there is an issue with root squashing on Solaris: root gets mapped to the user `noone`, which is not the same as the user `nobody`. If you are having trouble with file permissions as root on the client machine, be sure to check that the mapping works as you expect.

8.6.2. Solaris Clients

Solaris clients will regularly produce the following message:

```
svc: unknown program 100227 (me 100003)
```

This happens because Solaris clients, when they mount, try to obtain ACL information - which Linux obviously does not have. The messages can safely be ignored.

There are two known issues with diskless Solaris clients: First, a kernel version of at least 2.2.19 is needed to get `/dev/null` to export correctly. Second, the packet size may need to be set extremely small (i.e., 1024) on diskless sparc clients because the clients do not know how to assemble packets in reverse order. This can be done from `/etc/bootparams` on the clients.

8.7. SunOS

SunOS only has NFS Version 2 over UDP.

8.7.1. SunOS Servers

On the server end, SunOS uses the most traditional format for its `/etc/exports` file. The example in Section 3 would look like:

```
/usr    -access=slave1.foo.com,slave2.foo.com
/home   -rw=slave1.foo.com,slave2.foo.com, root=slave1.foo.com,slave2.foo.com
```

Again, the **`root`** option is listed for informational purposes and is not recommended unless necessary.

8.7.2. SunOS Clients

Be advised that SunOS makes all NFS locking requests as `daemon`, and therefore you will need to add the **`insecure_locks`** option to any volumes you export to a SunOS machine. See the **exports** man page for details.