Performance Tuning on Linux — NFS

Tune NFS Performance

**NFS performance is achieved through first tuning the underlying networking — Ethernet and TCP — and then selecting appropriate NFS parameters.** NFS was originally designed for trusted environments of a scale that today seems rather modest. The primary design goal was data integrity, not performance, which makes sense as Ethernet of the time ran at 10 Mbps and files and file systems were relatively small. The integrity goal led to a use of synchronous write operations, greatly slowing things down. But simultaneously, there was no lock function in the NFS server itself and multiple clients could write to the same file at the same time, leading to unpredictable corruption.

As for the trust built into the system, there was no real security mechanism beyond access control based on IP addresses plus the assumption that user and group IDs were consistent across all systems on the network.

The NFS server and the added NFS lock server were not required to run on any specific port, so the RPC port mapper was invented. The port mapper did a great job of solving a problem that didn't need to exist. It would start as one of the first network services, then all these other services expected to be listening on unpredictable ports would start and register themselves with the port mapper. *"If anyone asks, I'm listening on TCP port 2049."* A client would first connect to the port mapper on its predictable TCP port 111 and then ask it how to connect to the service they really wanted.

Like I said, a solution to a problem that shouldn't have been there in the first place. The NFS server was about the only network service that needed the port mapper, and it generally listened on the same port every time.

Evolution of NFS Versions

**NFS version 2** only allowed access to the first 2 GB of a file. The client made the decision about whether the user could access the file or not. This version supported UDP only, and it limited write operations to 8 kbytes. Ugh. It was a nice start, though.

**NFS version 3** supported much larger files. The client could ask the server to make the access decision — although it didn't have to and so we still had to fully trust all client hosts. NFSv3 over UDP could use up to 56 kbyte transactions. There was now support for NFS over TCP, although client implementations varied quite a bit as to details with associated compatibility issues.

**NFS version 4** added statefulness, so an NFSv4 client could directly inform the server about its locking, reading, writing the file, and so on. All the NFS application protocols — mount, stat, ACL checking, and the NFS file service itself, run over a single network protocol always running on a predictable TCP port 2049. There is no longer any need for an RPC port mapper to start first, or for separate mount and file locking daemons to run independently. Access control is now done by user name, not possibly inconsistent UID and GID as before, and security could be further enhanced with the use of Kerberos.

**NFS version 4.1** adds Parallel NFS (or pNFS), which can stripe data across multiple NFS servers. You can do block-level access and use NFS v4.1 much like Fibre Channel and iSCSI, and object access is meant to be analogous to AWS S3. NFS v4.1 adds some performance enhancements Perhaps more importantly to many current users, NFS v4.1 on Linux better inter-operates with non-Linux NFS servers and clients. Specifically, mounting NFS from Windows servers using Kerberos V5. For all the details see [RFC 5661](https://tools.ietf.org/html/rfc5661).

NFS Performance Goals

Rotating SATA and SAS disks can provide I/O of about 1 Gbps. Solid-state disks can fill the 6 Gbps bandwidth of a SATA 3 controller. NFS should be about this fast, filling a 1 Gbps LAN. NFS SAN appliances can fill a 10 Gbps LAN. Random I/O performance should be good, about 100 transactions per second per spindle.

Configuring NFS

On the server, specify what you will export, to which clients, and how, in /etc/exports.

On the client, specify what you will mount, from which server, and how, in [/etc/fstab](http://linux.die.net/man/5/fstab) for static mounts always made when the system starts, or through the [automounter](http://linux.die.net/man/8/automount) for mounts accessed only as needed. See the automounter details below.

NFS Performance

First, of course, optimize the local I/O on the server, both [block device I/O](https://cromwell-intl.com/open-source/performance-tuning/disks.html) and the [file systems](https://cromwell-intl.com/open-source/performance-tuning/file-systems.html). Then optimize both [Ethernet](https://cromwell-intl.com/open-source/performance-tuning/ethernet.html) and [TCP](https://cromwell-intl.com/open-source/performance-tuning/tcp.html) performance on the servers and clients. Only after doing that will it make sense to worry about NFS details.

Tuning the NFS Client

You can get NFS statistics for each mounted file system with mountstats.

# mountstats /path/to/mountpoint

For each NFS operation you get a count of how many times it has happens plus statistics on those events: bytes per operations, execute time, retransmissions and timeouts, etc. RTT or Round-Trip Time in milliseconds should be checked for NFS mounts used by interactive applications.

Specify NFS v4.1

Make sure to specify the appropriate NFS version. **You will have to experiment, this has been changing without corresponding updates in the manual pages or in Red Hat's on-line documentation.** You may find that both of the below work, but you may need to use just one or the other.

*possibly:*

# mount -o nfsvers=4.1 srvr:/export /mountpoint

*or possibly:*

# mount -o nfsvers=4,minorversion=1 srvr:/export /mountpoint

Specify Bytes Per Read/Write With rsize and wsize

The rsize and wsize options specify the number of bytes per NFS read and write request, respectively.

For the read size, set this as large as possible if your work load's reading is predominantly sequential — streaming media files, for example.

If your work load's reading is predominantly random, as for typical user file service, match the I/O size used on the server.

For the write size, you could try to match the I/O size on the server or simply maximize wsize. Try both and compare performance.

Both clients and servers should be able to do 1 MB read and write transfers.

Note that rsize and wsize are *requests,* the client and server must negotiate a mutually supported set of parameters. See the packet trace below.

Put options for static mounts in /etc/fstab. Put options for automounted file systems in the map file.

# grep nfs /etc/fstab

srvr1:/usr/local /usr/local nfs nfsvers=4.1,rsize=131072,wsize=131072 0 0

# tail -1 /etc/autofs/auto.master

/- auto.local

# cat /etc/autofs/auto.local

/usr/local -nfsvers=4.1,rsize=131072,wsize=131072 srvr1:/usr/local

What may be a better way is to use the systemd automounter.

# grep nfs /etc/fstab

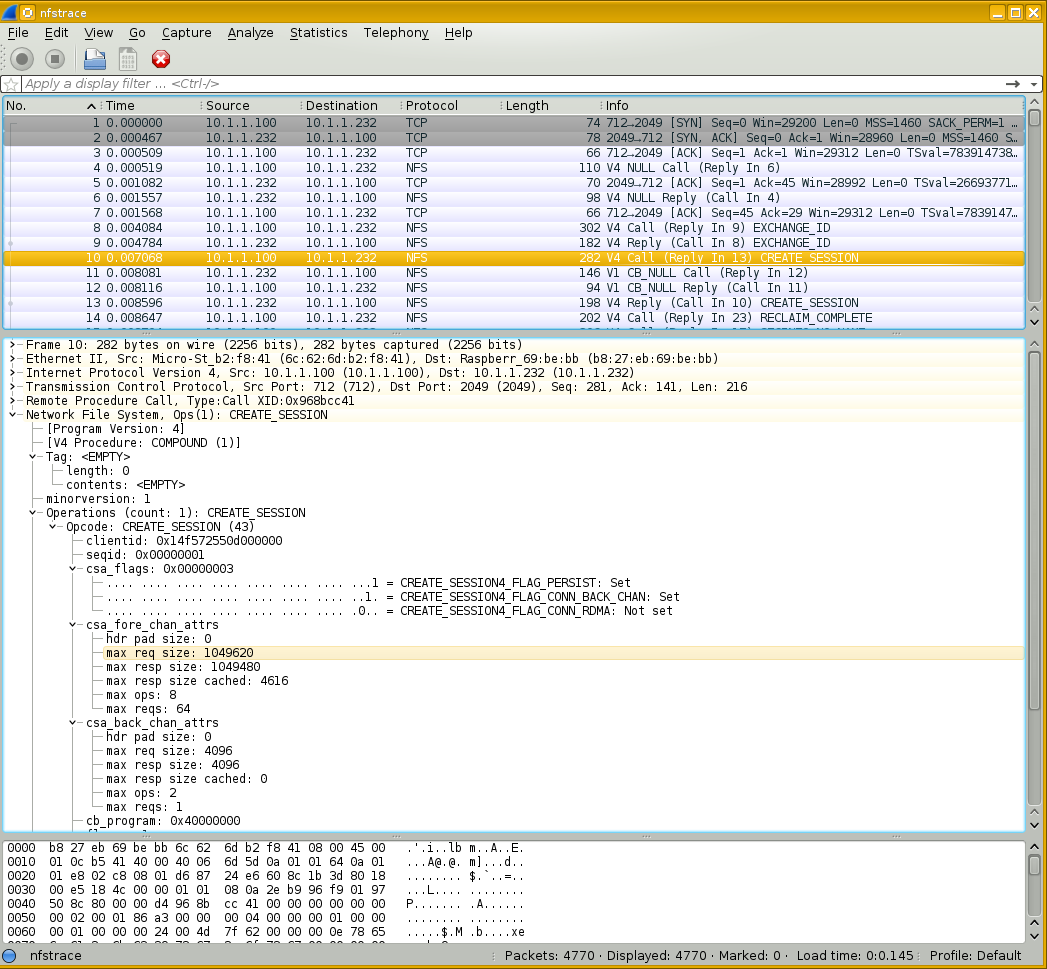
srvr1:/usr/local /usr/local nfs noauto,x-systemd.automount,nfsvers=4.1,rsize=131072,wsize=131072 0 0

Why Does The Output of mount Show Smaller rsize and wsize than I Requested?

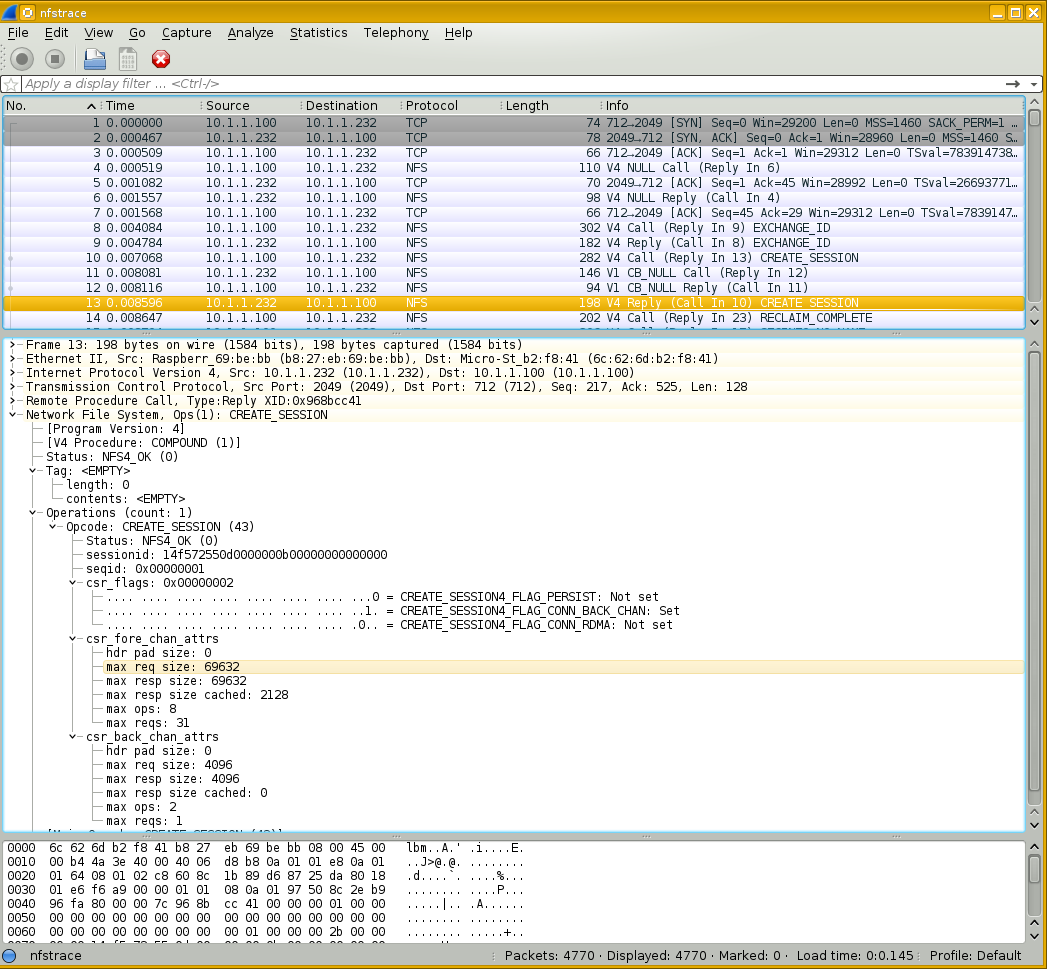
Short answer: Because what you got is all the server supports.

Detailed answer: Below is a Wireshark trace of an NFS 4.1 session negotiation. The selected packet shows the client request. See [section 17.36.5 of the IETF NFSv4.1 draft](https://tools.ietf.org/html/draft-ietf-nfsv4-minorversion1-10#page-395) for an explanation. Fields csa\_fore\_chan\_attrs and csa\_back\_chan\_attrs apply to attributes of the fore channel (also called the operations channel, requests from the client to server) and the back channel (requests from the server to client). Values ca\_maxrequestsize and ca\_maxresponsesize are the maximum sizes of requests that will be sent and responses that will be accepted. Wireshark displays these as max\_req\_size and max\_resp\_size.

Notice that the client 10.1.1.100 asks for 1,049,620. That is the maximum possible NFS payload, 1,048,576, plus 1,044.



This later packet is a response to the above one. Notice that the server 10.1.1.232 specifies max\_req\_size and max\_resp\_size of 69,632, and 4,096 + 65,536 = 69,632.



Run mount | grep nfs on the client and you will see rsize=65536,wsize=65536.

Tuning the NFS Server

First, apply all the earlier tuning to the local file system. If the server can't get data on and off its disks quickly, there's no hope of then getting that on and off the network quickly.

**Start plenty of NFS daemon threads.** The default is 8, increase this with RPCNFSDCOUNT=*NN* in /etc/sysconfig/nfs. Start *at least* one thread per processor core. There is little penalty in starting too many threads, make sure you have enough.

Check the NFS server statistics with nfsstat.

# nfsstat -4 -s

Start by tuning whatever your server does most.

* If read dominates, add RAM on the clients to cache more of the file systems and reduce read operations.
* If write dominates, make sure the clients are using noatime,nodiratime to avoid updating access times. Make sure your server's local file system access has been optimized.
* If getattr dominates, tune the attribute caches. If you were using noac (which many recommend against), try removing that.
* If readdir dominates, turn off attribute caching on the clients.

Examine kernel structure /proc/net/rpc/nfsd to see how busy the NFS server threads are. The line starting th lists the number of threads, and the last ten numbers are a histogram of the number of seconds the first 10% of threads were busy, the second 10%, and so on.

Load your system heavily and examine /proc/net/rpc/nfsd. Add threads until the last two numbers are zero or nearly zero. That means you have enough NFS server threads that they never or almost never are over 80% busy.

RDMA, or Remote Direct Memory Access

DMA or Direct Memory Access is a mechanism to allow an application to more directly read and write from and to local hardware, with fewer operating systems modules and buffers along the way.

RDMA extends that across a LAN, where an Ethernet packet arrives with RDMA extension. The application communicates directly with the Ethernet adapter. Application DMA access to the RDMA-capable Ethernet card is much like reaching across the LAN to the remote Ethernet adapter and through it to its application. This decreases latency and increases throughput (a rare case of making both better at once) and it decreases the CPU load at both ends. RDMA requires recent server-class Ethernet adapters, a recent kernel, and user-space tools.

Security and Reliability Issues

/etc/exports Syntax

First, be aware that /etc/exports is very fussy about its format. This does what you expect, sharing file system tree to clienthost in read-write mode:

/path/to/export clienthost(rw)

The space in the following means that there are *two* grantees of access to the exported file system. It will first grant access to clienthost, read-write because that's the default. *Then it also grants access to all other hosts in read-write mode because of the separate (rw).* This is almost certainly *not* what the administrator intended!

/path/to/export clienthost (rw)

root Squashing

The special user root is "squashed" to the user nobody in order to compartmentalize privileged access across an organization. A user might boot a system from media and be root on that workstation, change its IP if needed to gain access, and mount a file system. *But the user root and only that user is "squashed" to the unprivileged nobody and only has the privileges that are given to all users.*

If you do want root to have the usual full access on NFS-mounted file systems, export them with the no\_root\_squash option:

/path/to/export 10.0.0.0/8(rw,no\_root\_squash)

Subtree Checking

Let's say that you export /usr/local, which is part of the root file system.

# cat /etc/exports

/usr/local 10.0.0.0/8(rw,no\_root\_squash)

# df /usr/local

Filesystem Size Used Avail Use% Mounted on

/dev/sda3 150G 68G 67G 45% /

There is a risk that an illicit administrator, possibly on a compromised client, could guess the file handle for a file on that file system but not contained within the /usr/local hierarchy. As the export does not squash root, the file /etc/shadow would be of particular interest on that file system but outside the export.

There is an option subtree\_check which enforces additional checks on the server to verify that the requested file is contained within the exported hierarchy. You can specify subtree\_check to force this security check or no\_subtree\_check to avoid it. The default behavior has changed back and forth. It is easy to check which mounted file system a file is in, it is surprisingly expensive to check if it is contained within the exported hierarchy within that.

It has lately been felt that the minor security improvement provided by subtree\_check is overwhelmed by the associated performance degradation, so from release 1.0.0 of the nfs-utils package forward, no\_subtree\_check is the default.

**The best solution is to make an exported hierarchy be an independent file system.** In our example, if we want to share /usr/local via NFS, then it should be an independent file system.

# cat /etc/exports

/usr/local 10.0.0.0/8(rw,no\_root\_squash)

# df /usr/local

Filesystem Size Used Avail Use% Mounted on

/dev/sdb1 917G 565G 353G 62% /usr/local

And next...

We have finished tuning the Linux operating system! **Now we can** [**measure application resource use**](https://cromwell-intl.com/open-source/performance-tuning/applications.html) **to see if we need to make any changes.**