NAME

nfs - fstab format and options for the **nfs** file systems

SYNOPSIS

/etc/fstab

DESCRIPTION

NFS is an Internet Standard protocol created by Sun Microsystems in 1984. NFS was developed to allow file sharing between systems residing on a local area network. The Linux NFS client supports three versions of the NFS protocol: NFS version 2 [RFC1094], NFS version 3 [RFC1813], and NFS version 4 [RFC3530].

The mount(8) command attaches a file system to the system's name space hierarchy at a given mount point. The*/etc/fstab* file describes ho w mount(8) should assemble a system's file name hierarchy from various independent file systems (including file systems exported by NFS servers). Each line in the */etc/fstab* file describes a single file system, its mount point, and a set of default mount options for that mount point.

For NFS file system mounts, a line in the */etc/fstab* file specifies the server name, the path name of the exported server directory to mount, the local directory that is the mount point, the type of file system that is being mounted, and a list of mount options that control the way the filesystem is mounted and how the NFS client behaves when accessing files on this mount point. The fifth and sixth fields on each line are not used by NFS, thus conventionally each contain the digit zero. For example:

server:path /mountpoint fstype option,option,... 0 0

The server's hostname and export pathname are separated by a colon, while the mount options are separated by commas. The remaining fields are separated by blanks or tabs.

The server's hostname can be an unqualified hostname, a fully qualified domain name, a dotted quad IPv4 address, or an IPv6 address enclosed in square brackets. Link-local and site-local IPv6 addresses must be accompanied by an interface identifier. See ipv6(7) for details on specifying raw IPv6 addresses.

The *fstype* field contains nfs. Use of the nfs4 fstype in */etc/fstab* is deprecated.

MOUNT OPTIONS

Refer to mount(8) for a description of generic mount options available for all file systems. If you do not need to specify any mount options, use the generic option **defaults** in */etc/fstab*.

Options supported by all versions

These options are valid to use with any NFS version.

soft / **hard**
: Determines the recovery behavior of the NFS client after an NFS request times out. If neither option is specified (or if the **hard** option is specified), NFS requests are retried indefinitely. If the**soft** option is specified, then the NFS client fails an NFS request after **retrans** retransmissions have been sent, causing the NFS client to return an error to the calling application.

  NB: A so-called soft timeout can cause silent data corruption in certain cases. As such, use the **soft** option only when client responsiveness is more important than data integrity. Using NFS over TCP or increasing the value of the **retrans** option may mitigate some of the risks of using the **soft** option.

timeo=*n*
: The time in deciseconds (tenths of a second) the NFS client waits for a response before it retries an NFS request.

  For NFS over TCP the default **timeo** value is 600 (60 seconds). The NFS client performs linear backoff: After each retransmission the timeout is increased by **timeo** up to the maximum of 600 seconds.

However, for NFS over UDP, the client uses an adaptive algorithm to estimate an appropriate timeout value for frequently used request types (such as READ and WRITE requests), but uses the **timeo** setting for infrequently used request types (such as FSINFO requests). If the **timeo** option is not specified, infrequently used request types are retried after 1.1 seconds. After each retransmission, the NFS client doubles the timeout for that request, up to a maximum timeout length of 60 seconds.

retrans=*n*

The number of times the NFS client retries a request before it attempts further recovery action. If the **retrans** option is not specified, the NFS client tries each request three times.

The NFS client generates a server not responding message after **retrans** retries, then attempts further recovery (depending on whether the **hard** mount option is in effect).

rsize=*n*

The maximum number of bytes in each network READ request that the NFS client can receive when reading data from a file on an NFS server. The actual data payload size of each NFS READ request is equal to or smaller than the **rsize** setting. The largest read payload supported by the Linux NFS client is 1,048,576 bytes (one megabyte).

The **rsize** value is a positive integral multiple of 1024. Specified **rsize** values lower than 1024 are replaced with 4096; values larger than 1048576 are replaced with 1048576. If a specified value is within the supported range but not a multiple of 1024, it is rounded down to the nearest multiple of 1024.

If an **rsize** value is not specified, or if the specified **rsize** value is larger than the maximum that either client or server can support, the client and server negotiate the largest **rsize** value that they can both support.

The **rsize** mount option as specified on the mount(8) command line appears in the */etc/mtab* file. However, the effective **rsize** value negotiated by the client and server is reported in the */proc/mounts* file.

wsize=*n*

The maximum number of bytes per network WRITE request that the NFS client can send when writing data to a file on an NFS server. The actual data payload size of each NFS WRITE request is equal to or smaller than the **wsize** setting. The largest write payload supported by the Linux NFS client is 1,048,576 bytes (one megabyte).

Similar to **rsize** , the **wsize** value is a positive integral multiple of 1024. Specified **wsize** values lower than 1024 are replaced with 4096; values larger than 1048576 are replaced with 1048576. If a specified value is within the supported range but not a multiple of 1024, it is rounded down to the nearest multiple of 1024.

If a **wsize** value is not specified, or if the specified **wsize** value is larger than the maximum that either client or server can support, the client and server negotiate the largest **wsize** value that they can both support.

The **wsize** mount option as specified on the mount(8) command line appears in the */etc/mtab* file. However, the effective **wsize** value negotiated by the client and server is reported in the */proc/mounts* file.

ac / **noac**

Selects whether the client may cache file attributes. If neither option is specified (or if **ac** is specified), the client caches file attributes.

To improve performance, NFS clients cache file attributes. Every few seconds, an NFS client checks the server's version of each file's attributes for updates. Changes that occur on the server in those small intervals remain

undetected until the client checks the server again. The **noac** option prevents clients from caching file attributes so that applications can more quickly detect file changes on the server.

In addition to preventing the client from caching file attributes, the **noac** option forces application writes to become synchronous so that local changes to a file become visible on the server immediately. That way, other clients can quickly detect recent writes when they check the file's attributes.

Using the **noac** option provides greater cache coherence among NFS clients accessing the same files, but it extracts a significant performance penalty. As such, judicious use of file locking is encouraged instead. The DATA AND METADATA COHERENCE section contains a detailed discussion of these trade-offs.

acregmin=*n* The minimum time (in seconds) that the NFS client caches attributes of a regular file before it requests fresh attribute information from a server. If this option is not specified, the NFS client uses a 3-second minimum.

acregmax=*n* The maximum time (in seconds) that the NFS client caches attributes of a regular file before it requests fresh attribute information from a server. If this option is not specified, the NFS client uses a 60-second maximum.

acdirmin=*n* The minimum time (in seconds) that the NFS client caches attributes of a directory before it requests fresh attribute information from a server. If this option is not specified, the NFS client uses a 30-second minimum.

acdirmax=*n* The maximum time (in seconds) that the NFS client caches attributes of a directory before it requests fresh attribute information from a server. If this option is not specified, the NFS client uses a 60-second maximum.

actimeo=*n* Using **actimeo** sets all of **acregmin**, **acregmax**, **acdirmin**, and **acdirmax** to the same value. If this option is not specified, the NFS client uses the defaults for each of these options listed above.

bg / **fg** Determines how the mount(8) command behaves if an attempt to mount an export fails. The **fg** option causes mount(8) to exit with an error status if any part of the mount request times out or fails outright. This is called a foreground mount, and is the default behavior if neither the **fg** nor **bg** mount option is specified.

If the **bg** option is specified, a timeout or failure causes the mount(8) command to fork a child which continues to attempt to mount the export. The parent immediately returns with a zero exit code. This is known as a background mount.

If the local mount point directory is missing, the mount(8) command acts as if the mount request timed out. This permits nested NFS mounts specified in */etc/fstab* to proceed in any order during system initialization, even if some NFS servers are not yet available. Alternatively these issues can be addressed using an automounter (refer to **automount(8)** for details).

rdirplus / **nordirplus** Selects whether to use NFS v3 or v4 READDIRPLUS requests. If this option is not specified, the NFS client uses READDIRPLUS requests on NFS v3 or v4 mounts to read small directories. Some applications perform better if the client uses only READDIR requests for all directories.

retry=*n* The number of minutes that the mount(8) command retries an NFS mount operation in the foreground or background before giving up. If this option is not specified, the default value for foreground mounts is 2

minutes, and the default value for background mounts is 10000 minutes (80 minutes shy of one week). If a value of zero is specified, the mount(8) command exits immediately after the first failure.

sec=*flavor*
: The security flavor to use for accessing files on this mount point. If the server does not support this flavor, the mount operation fails. If **sec=** is not specified, the client attempts to find a security flavor that both the client and the server supports. Valid *flavors* are **none**, **sys**, **krb5**, **krb5i**, and **krb5p**. Refer to the SECURITY CONSIDERATIONS section for details.

sharecache / **nosharecache**
: Determines how the client's data cache and attribute cache are shared when mounting the same export more than once concurrently. Using the same cache reduces memory requirements on the client and presents identical file contents to applications when the same remote file is accessed via different mount points.

  If neither option is specified, or if the **sharecache** option is specified, then a single cache is used for all mount points that access the same export. If the **nosharecache** option is specified, then that mount point gets a unique cache. Note that when data and attribute caches are shared, the mount options from the first mount point take effect for subsequent concurrent mounts of the same export.

  As of kernel 2.6.18, the behavior specified by **nosharecache** is legacy caching behavior. This is considered a data risk since multiple cached copies of the same file on the same client can become out of sync following a local update of one of the copies.

resvport / **noresvport**
: Specifies whether the NFS client should use a privileged source port when communicating with an NFS server for this mount point. If this option is not specified, or the **resvport** option is specified, the NFS client uses a privileged source port. If the **noresvport** option is specified, the NFS client uses a non-privileged source port. This option is supported in kernels 2.6.28 and later.

  Using non-privileged source ports helps increase the maximum number of NFS mount points allowed on a client, but NFS servers must be configured to allow clients to connect via non-privileged source ports.

  Refer to the SECURITY CONSIDERATIONS section for important details.

lookupcache=*mode*
: Specifies how the kernel manages its cache of directory entries for a given mount point. *mode* can be one of **all**, **none**, **pos**, or **positive**. This option is supported in kernels 2.6.28 and later.

  The Linux NFS client caches the result of all NFS LOOKUP requests. If the requested directory entry exists on the server, the result is referred to as *positive*. If the requested directory entry does not exist on the server, the result is referred to as *negative*.

  If this option is not specified, or if **all** is specified, the client assumes both types of directory cache entries are valid until their parent directory's cached attributes expire.

  If **pos** or **positive** is specified, the client assumes positive entries are valid until their parent directory's cached attributes expire, but always revalidates negative entires before an application can use them.

If **none** is specified, the client revalidates both types of directory cache entries before an application can use them. This permits quick detection of files that were created or removed by other clients, but can impact application and server performance.

The DATA AND METADATA COHERENCE section contains a detailed discussion of these trade-offs.

fsc / nofsc Enable/Disables the cache of (read-only) data pages to the local disk using the FS-Cache facility. See cachefilesd(8) and <kernel_soruce>/Documentation/filesystems/caching for detail on how to configure the FS-Cache facility. Default value is nofsc.

Options for NFS versions 2 and 3 only

Use these options, along with the options in the above subsection, for NFS versions 2 and 3 only.

proto=*netid* The *netid* determines the transport that is used to communicate with the NFS server. Available options are **udp**, **udp6**, **tcp**, **tcp6**, and **rdma**. Those which end in **6** use IPv6 addresses and are only available if support for TI-RPC is built in. Others use IPv4 addresses.

Each transport protocol uses different default **retrans** and **timeo** settings. Refer to the description of these two mount options for details.

In addition to controlling how the NFS client transmits requests to the server, this mount option also controls how the mount(8) command communicates with the server's rpcbind and mountd services. Specifying a netid that uses TCP forces all traffic from the mount(8) command and the NFS client to use TCP. Specifying a netid that uses UDP forces all traffic types to use UDP.

Before using NFS over UDP, refer to the TRANSPORT METHODS section.

If the **proto** mount option is not specified, the mount(8) command discovers which protocols the server supports and chooses an appropriate transport for each service. Refer to the TRANSPORT METHODS section for more details.

udp The **udp** option is an alternative to specifying **proto=udp.** It is included for compatibility with other operating systems.

Before using NFS over UDP, refer to the TRANSPORT METHODS section.

tcp The **tcp** option is an alternative to specifying **proto=tcp.** It is included for compatibility with other operating systems.

rdma The **rdma** option is an alternative to specifying **proto=rdma.**

port=*n* The numeric value of the server's NFS service port. If the server's NFS service is not available on the specified port, the mount request fails.

If this option is not specified, or if the specified port value is 0, then the NFS client uses the NFS service port number advertised by the server's rpcbind service. The mount request fails if the server's rpcbind service is not available, the server's NFS service is not registered with its rpcbind service, or the server's NFS service is not available on the advertised port.

mountport=*n* The numeric value of the server's mountd port. If the server's mountd service is not available on the specified port, the mount request fails.

If this option is not specified, or if the specified port value is 0, then the mount(8) command uses the mountd service port number advertised by the server's rpcbind service. The mount request fails if the server's rpcbind service is not available, the server's mountd service is not registered with its rpcbind service, or the server's mountd service is not available on the advertised port.

This option can be used when mounting an NFS server through a firewall that blocks the rpcbind protocol.

mountproto=*netid* The transport the NFS client uses to transmit requests to the NFS server's mountd service when performing this mount request, and when later unmounting this mount point.

netid may be one of **udp**, and **tcp** which use IPv4 address or, if TI-RPC is built into the **mount.nfs** command, **udp6**, and **tcp6** which use IPv6 addresses.

This option can be used when mounting an NFS server through a firewall that blocks a particular transport. When used in combination with the **proto** option, different transports for mountd requests and NFS requests can be specified. If the server's mountd service is not available via the specified transport, the mount request fails.

Refer to the TRANSPORT METHODS section for more on how the **mountproto** mount option interacts with the **proto** mount option.

mounthost=*name* The hostname of the host running mountd. If this option is not specified, the mount(8) command assumes that the mountd service runs on the same host as the NFS service.

mountvers=*n* The RPC version number used to contact the server's mountd. If this option is not specified, the client uses a version number appropriate to the requested NFS version. This option is useful when multiple NFS services are running on the same remote server host.

namlen=*n* The maximum length of a pathname component on this mount. If this option is not specified, the maximum length is negotiated with the server. In most cases, this maximum length is 255 characters.

Some early versions of NFS did not support this negotiation. Using this option ensures that pathconf(3) reports the proper maximum component length to applications in such cases.

nfsvers=*n* The NFS protocol version number used to contact the server's NFS service. If the server does not support the requested version, the mount request fails. If this option is not specified, the client negotiates a suitable version with the server, trying version 4 first, version 3 second, and version 2 last.

vers=*n* This option is an alternative to the **nfsvers** option. It is included for compatibility with other operating systems.

lock / **nolock** Selects whether to use the NLM sideband protocol to lock files on the server. If neither option is specified (or if**loc k** is specified), NLM locking is used for this mount point. When using the**noloc k** option, applications can lock files, but such locks provide exclusion only against other applications running on the same client. Remote applications are not affected by these locks.

NLM locking must be disabled with the **nolock** option when using NFS to mount */var* because */var* contains files used by the NLM

implementation on Linux. Using the**noloc k** option is also required when mounting exports on NFS servers that do not support the NLM protocol.

intr / **nointr** Selects whether to allow signals to interrupt file operations on this mount point. If neither option is specified (or if **nointr** is specified), signals do not interrupt NFS file operations. If **intr** is specified, system calls return EINTR if an in-progress NFS operation is interrupted by a signal.

Using the **intr** option is preferred to using the **soft** option because it is significantly less likely to result in data corruption.

The **intr** / **nointr** mount option is deprecated after kernel 2.6.25. Only SIGKILL can interrupt a pending NFS operation on these kernels, and if specified, this mount option is ignored to provide backwards compatibility with older kernels.

cto / **nocto** Selects whether to use close-to-open cache coherence semantics. If neither option is specified (or if **cto** is specified), the client uses close-to-open cache coherence semantics. If the **nocto** option is specified, the client uses a non-standard heuristic to determine when files on the server have changed.

Using the **nocto** option may improve performance for read-only mounts, but should be used only if the data on the server changes only occasionally. The DATA AND METADATA COHERENCE section discusses the behavior of this option in more detail.

acl / **noacl** Selects whether to use the NFSACL sideband protocol on this mount point. The NFSACL sideband protocol is a proprietary protocol implemented in Solaris that manages Access Control Lists. NFSACL was never made a standard part of the NFS protocol specification.

If neither **acl** nor **noacl** option is specified, the NFS client negotiates with the server to see if the NFSACL protocol is supported, and uses it if the server supports it. Disabling the NFSACL sideband protocol may be necessary if the negotiation causes problems on the client or server. Refer to the SECURITY CONSIDERATIONS section for more details.

local_lock=mechanism

Specifies whether to use local locking for any or both of the flock and the POSIX locking mechanisms. *mechanism* can be one of **all**, **flock**, **posix**, or **none**. This option is supported in kernels 2.6.37 and later.

The Linux NFS client provides a way to make locks local. This means, the applications can lock files, but such locks provide exclusion only against other applications running on the same client. Remote applications are not affected by these locks.

If this option is not specified, or if **none** is specified, the client assumes that the locks are not local.

If **all** is specified, the client assumes that both flock and POSIX locks are local.

If **flock** is specified, the client assumes that only flock locks are local and uses NLM sideband protocol to lock files when POSIX locks are used.

If **posix** is specified, the client assumes that POSIX locks are local and uses NLM sideband protocol to lock files when flock locks are used.

To support legacy flock behavior similar to that of NFS clients < 2.6.12, use 'local_lock=flock'. This option is required when exporting NFS

mounts via Samba as Samba maps Windows share mode locks as flock. Since NFS clients > 2.6.12 implement flock by emulating POSIX locks, this will result in conflicting locks.

NOTE: When used together, the 'local_lock' mount option will be overridden by 'nolock'/'lock' mount option.

Options for NFS version 4 only

Use these options, along with the options in the first subsection above, for NFS version 4 and newer.

proto=*netid*

The *netid* determines the transport that is used to communicate with the NFS server. Supported options are **tcp**, **tcp6**, and **rdma**. **tcp6** use IPv6 addresses and is only available if support for TI-RPC is built in. Both others use IPv4 addresses.

All NFS version 4 servers are required to support TCP, so if this mount option is not specified, the NFS version 4 client uses the TCP protocol. Refer to the TRANSPORT METHODS section for more details.

port=*n*

The numeric value of the server's NFS service port. If the server's NFS service is not available on the specified port, the mount request fails.

If this mount option is not specified, the NFS client uses the standard NFS port number of 2049 without first checking the server's rpcbind service. This allows an NFS version 4 client to contact an NFS version 4 server through a firewall that may block rpcbind requests.

If the specified port value is 0, then the NFS client uses the NFS service port number advertised by the server's rpcbind service. The mount request fails if the server's rpcbind service is not available, the server's NFS service is not registered with its rpcbind service, or the server's NFS service is not available on the advertised port.

intr / **nointr**

Selects whether to allow signals to interrupt file operations on this mount point. If neither option is specified (or if **intr** is specified), system calls return EINTR if an in-progress NFS operation is interrupted by a signal. If **nointr** is specified, signals do not interrupt NFS operations.

Using the **intr** option is preferred to using the **soft** option because it is significantly less likely to result in data corruption.

The **intr** / **nointr** mount option is deprecated after kernel 2.6.25. Only SIGKILL can interrupt a pending NFS operation on these kernels, and if specified, this mount option is ignored to provide backwards compatibility with older kernels.

cto / **nocto**

Selects whether to use close-to-open cache coherence semantics for NFS directories on this mount point. If neither**cto** nor **nocto** is specified, the default is to use close-to-open cache coherence semantics for directories.

File data caching behavior is not affected by this option. The DATA AND METADATA COHERENCE section discusses the behavior of this option in more detail.

clientaddr=*n.n.n.n*

clientaddr=*n:n:...:n*

Specifies a single IPv4 address (in dotted-quad form), or a non-link-local IPv6 address, that the NFS client advertises to allow servers to perform NFS version 4 callback requests against files on this mount point. If the server is unable to establish callback connections to clients, performance may degrade, or accesses to files may temporarily hang.

If this option is not specified, the mount(8) command attempts to discover an appropriate callback address automatically. The automatic discovery process is not perfect, however. In the presence of multiple client network interfaces, special routing policies, or atypical network topologies, the exact address to use for callbacks may be nontrivial to determine.

nfs4 FILE SYSTEM TYPE

The **nfs4** file system type is an old syntax for specifying NFSv4 usage. It can still be used with all NFSv4-specific and common options, excepted the **nfsvers** mount option.

MOUNT CONFIGURATION FILE

If the mount command is configured to do so, all of the mount options described in the previous section can also be configured in the */etc/nfsmount.conf* file. See **nfsmount.conf(5)** for details.

EXAMPLES

To mount an export using NFS version 2, use the **nfs** file system type and specify the **nfsvers=2** mount option. To mount using NFS version 3, use the **nfs** file system type and specify the **nfsvers=3** mount option. To mount using NFS version 4, use either the **nfs** file system type, with the **nfsvers=4** mount option, or the **nfs4** file system type.

The following example from an */etc/fstab* file causes the mount command to negotiate reasonable defaults for NFS behavior.

server:/export /mnt nfs defaults 0 0

Here is an example from an /etc/fstab file for an NFS version 2 mount over UDP.

server:/export /mnt nfs nfsvers=2,proto=udp 0 0

Try this example to mount using NFS version 4 over TCP with Kerberos 5 mutual authentication.

server:/export /mnt nfs4 sec=krb5 0 0

This example can be used to mount /usr over NFS.

server:/export /usr nfs ro,nolock,nocto,actimeo=3600 0 0

This example shows how to mount an NFS server using a raw IPv6 link-local address.

[fe80::215:c5ff:fb3e:e2b1%eth0]:/export /mnt nfs defaults 0 0

TRANSPORT METHODS

NFS clients send requests to NFS servers via Remote Procedure Calls, or *RPCs*. The RPC client discovers remote service endpoints automatically, handles per-request authentication, adjusts request parameters for different byte endianness on client and server, and retransmits requests that may have been lost by the network or server. RPC requests and replies flow over a network transport.

In most cases, the mount(8) command, NFS client, and NFS server can automatically negotiate proper transport and data transfer size settings for a mount point. In some cases, however, it pays to specify these settings explicitly using mount options.

Traditionally, NFS clients used the UDP transport exclusively for transmitting requests to servers. Though its implementation is simple, NFS over UDP has many limitations that prevent smooth operation and good performance in some common deployment environments. Even an insignificant packet loss rate results in the loss of whole NFS requests; as such, retransmit timeouts are usually in the subsecond range to allow clients to recover quickly from dropped requests, but this can result in extraneous network traffic and server load.

However, UDP can be quite effective in specialized settings where the networks MTU is large relative to NFSs data transfer size (such as network environments that enable jumbo Ethernet frames). In such environments, trimming the **rsize** and **wsize** settings so that each NFS read or write request fits in just a few network frames (or even in a single frame) is advised. This reduces the probability that the loss of a single MTU-sized network frame results in the loss of an entire

large read or write request.

TCP is the default transport protocol used for all modern NFS implementations. It performs well in almost every conceivable network environment and provides excellent guarantees against data corruption caused by network unreliability. TCP is often a requirement for mounting a server through a network firewall.

Under normal circumstances, networks drop packets much more frequently than NFS servers drop requests. As such, an aggressive retransmit timeout setting for NFS over TCP is unnecessary. Typical timeout settings for NFS over TCP are between one and ten minutes. After the client exhausts its retransmits (the value of the **retrans** mount option), it assumes a network partition has occurred, and attempts to reconnect to the server on a fresh socket. Since TCP itself makes network data transfer reliable, **rsize** and **wsize** can safely be allowed to default to the largest values supported by both client and server, independent of the network's MTU size.

Using the mountproto mount option

This section applies only to NFS version 2 and version 3 mounts since NFS version 4 does not use a separate protocol for mount requests.

The Linux NFS client can use a different transport for contacting an NFS server's rpcbind service, its mountd service, its Network Lock Manager (NLM) service, and its NFS service. The exact transports employed by the Linux NFS client for each mount point depends on the settings of the transport mount options, which include **proto**, **mountproto**, **udp**, and **tcp**.

The client sends Network Status Manager (NSM) notifications via UDP no matter what transport options are specified, but listens for server NSM notifications on both UDP and TCP. The NFS Access Control List (NFSACL) protocol shares the same transport as the main NFS service.

If no transport options are specified, the Linux NFS client uses UDP to contact the server's mountd service, and TCP to contact its NLM and NFS services by default.

If the server does not support these transports for these services, the mount(8) command attempts to discover what the server supports, and then retries the mount request once using the discovered transports. If the server does not advertise any transport supported by the client or is misconfigured, the mount request fails. If the **bg** option is in effect, the mount command backgrounds itself and continues to attempt the specified mount request.

When the **proto** option, the **udp** option, or the **tcp** option is specified but the **mountproto** option is not, the specified transport is used to contact both the server's mountd service and for the NLM and NFS services.

If the **mountproto** option is specified but none of the **proto**, **udp** or **tcp** options are specified, then the specified transport is used for the initial mountd request, but the mount command attempts to discover what the server supports for the NFS protocol, preferring TCP if both transports are supported.

If both the **mountproto** and **proto** (or **udp** or **tcp**) options are specified, then the transport specified by the **mountproto** option is used for the initial mountd request, and the transport specified by the **proto** option (or the **udp** or **tcp** options) is used for NFS, no matter what order these options appear. No automatic service discovery is performed if these options are specified.

If any of the **proto**, **udp**, **tcp**, or **mountproto** options are specified more than once on the same mount command line, then the value of the rightmost instance of each of these options takes effect.

Using NFS over UDP on high-speed links

Using NFS over UDP on high-speed links such as Gigabit **can cause silent data corruption**.

The problem can be triggered at high loads, and is caused by problems in IP fragment reassembly. NFS read and writes typically transmit UDP packets of 4 Kilobytes or more, which have to be broken up into several fragments in order to be sent over the Ethernet link, which limits packets to 1500 bytes by default. This process happens at the IP network layer and is called

fragmentation.

In order to identify fragments that belong together, IP assigns a 16bit *IP ID* value to each packet; fragments generated from the same UDP packet will have the same IP ID. The receiving system will collect these fragments and combine them to form the original UDP packet. This process is called reassembly. The default timeout for packet reassembly is 30 seconds; if the network stack does not receive all fragments of a given packet within this interval, it assumes the missing fragment(s) got lost and discards those it already received.

The problem this creates over high-speed links is that it is possible to send more than 65536 packets within 30 seconds. In fact, with heavy NFS traffic one can observe that the IP IDs repeat after about 5 seconds.

This has serious effects on reassembly: if one fragment gets lost, another fragment *from a different packet* but with the *same IP ID* will arrive within the 30 second timeout, and the network stack will combine these fragments to form a new packet. Most of the time, network layers above IP will detect this mismatched reassembly - in the case of UDP, the UDP checksum, which is a 16 bit checksum over the entire packet payload, will usually not match, and UDP will discard the bad packet.

However, the UDP checksum is 16 bit only, so there is a chance of 1 in 65536 that it will match even if the packet payload is completely random (which very often isn't the case). If that is the case, silent data corruption will occur.

This potential should be taken seriously, at least on Gigabit Ethernet. Network speeds of 100Mbit/s should be considered less problematic, because with most traffic patterns IP ID wrap around will take much longer than 30 seconds.

It is therefore strongly recommended to use **NFS over TCP where possible**, since TCP does not perform fragmentation.

If you absolutely have to use NFS over UDP over Gigabit Ethernet, some steps can be taken to mitigate the problem and reduce the probability of corruption:

Jumbo frames:
: Many Gigabit network cards are capable of transmitting frames bigger than the 1500 byte limit of traditional Ethernet, typically 9000 bytes. Using jumbo frames of 9000 bytes will allow you to run NFS over UDP at a page size of 8K without fragmentation. Of course, this is only feasible if all involved stations support jumbo frames.

  To enable a machine to send jumbo frames on cards that support it, it is sufficient to configure the interface for a MTU value of 9000.

Lower reassembly timeout:
: By lowering this timeout below the time it takes the IP ID counter to wrap around, incorrect reassembly of fragments can be prevented as well. To do so, simply write the new timeout value (in seconds) to the file **/proc/sys/net/ipv4/ipfrag_time**.

  A value of 2 seconds will greatly reduce the probability of IPID clashes on a single Gigabit link, while still allowing for a reasonable timeout when receiving fragmented traffic from distant peers.

DATA AND METADATA COHERENCE

Some modern cluster file systems provide perfect cache coherence among their clients. Perfect cache coherence among disparate NFS clients is expensive to achieve, especially on wide area networks. As such, NFS settles for weaker cache coherence that satisfies the requirements of most file sharing types. Normally, file sharing is completely sequential: first client A opens a file, writes something to it, then closes it; then client B opens the same file, and reads the changes.

Close-to-open cache consistency

When an application opens a file stored on an NFS server, the NFS client checks that it still exists on the server and is permitted to the opener by sending a GETATTR or ACCESS request. When the application closes the file, the NFS client writes back any pending changes to the file so that the next opener can view the changes. This also gives the NFS client an opportunity to report any server write errors to the application via the return code from close(2). The behavior of checking at open time and flushing at close time is referred to as close-to-open cache consistency.

Weak cache consistency

There are still opportunities for a client's data cache to contain stale data. The NFS version 3 protocol introduced weak cache consistency (also known as WCC) which provides a way of efficiently checking a file's attributes before and after a single request. This allows a client to help identify changes that could have been made by other clients.

When a client is using many concurrent operations that update the same file at the same time (for example, during asynchronous write behind), it is still difficult to tell whether it was that client's updates or some other client's updates that altered the file.

Attribute caching

Use the **noac** mount option to achieve attribute cache coherence among multiple clients. Almost every file system operation checks file attribute information. The client keeps this information cached for a period of time to reduce network and server load. When **noac** is in effect, a client's file attribute cache is disabled, so each operation that needs to check a file's attributes is forced to go back to the server. This permits a client to see changes to a file very quickly, at the cost of many extra network operations.

Be careful not to confuse the **noac** option with no data caching. The**noac** moun t option prevents the client from caching file metadata, but there are still races that may result in data cache incoherence between client and server.

The NFS protocol is not designed to support true cluster file system cache coherence without some type of application serialization. If absolute cache coherence among clients is required, applications should use file locking. Alternatively, applications can also open their files with the O_DIRECT flag to disable data caching entirely.

Directory entry caching

The Linux NFS client caches the result of all NFS LOOKUP requests. If the requested directory entry exists on the server, the result is referred to as a *positive* lookup result. If the requested directory entry does not exist on the server (that is, the server returned ENOENT), the result is referred to as *negative* lookup result.

To detect when directory entries have been added or removed on the server, the Linux NFS client watches a directory's mtime. If the client detects a change in a directory's mtime, the client drops all cached LOOKUP results for that directory. Since the directory's mtime is a cached attribute, it may take some time before a client notices it has changed. See the descriptions of the **acdirmin**, **acdirmax**, and **noac** mount options for more information about how long a directory's mtime is cached.

Caching directory entries improves the performance of applications that do not share files with applications on other clients. Using cached information about directories can interfere with applications that run concurrently on multiple clients and need to detect the creation or removal of files quickly, however. The**lookupcac he** mount option allows some tuning of directory entry caching behavior.

Before kernel release 2.6.28, the Linux NFS client tracked only positive lookup results. This permitted applications to detect new directory entries created by other clients quickly while still providing some of the performance benefits of caching. If an application depends on the previous lookup caching behavior of the Linux NFS client, you can use **lookupcache=positive**.

If the client ignores its cache and validates every application lookup request with the server, that client can immediately detect when a new directory entry has been either created or removed by another client. You can specify this behavior using **lookupcache=none**. The extra NFS requests needed if the client does not cache directory entries can exact a performance penalty. Disabling lookup caching should result in less of a performance penalty than using **noac**, and has no effect on how the NFS client caches the attributes of files.

The sync mount option

The NFS client treats the **sync** mount option differently than some other file systems (refer to mount(8) for a description of the generic **sync** and **async** mount options). If neither **sync** nor **async** is specified (or if the **async** option is specified), the NFS client delays sending application writes to the server until any of these events occur:

Memory pressure forces reclamation of system memory resources.

An application flushes file data explicitly with sync(2), msync(2), or **fsync(3).**

An application closes a file with close(2).

The file is locked/unlocked via fcntl(2).

In other words, under normal circumstances, data written by an application may not immediately appear on the server that hosts the file.

If the **sync** option is specified on a mount point, any system call that writes data to files on that mount point causes that data to be flushed to the server before the system call returns control to user space. This provides greater data cache coherence among clients, but at a significant performance cost.

Applications can use the O_SYNC open flag to force application writes to individual files to go to the server immediately without the use of the **sync** mount option.

Using file locks with NFS

The Network Lock Manager protocol is a separate sideband protocol used to manage file locks in NFS version 2 and version 3. To support lock recovery after a client or server reboot, a second sideband protocol -- known as the Network Status Manager protocol -- is also required. In NFS version 4, file locking is supported directly in the main NFS protocol, and the NLM and NSM sideband protocols are not used.

In most cases, NLM and NSM services are started automatically, and no extra configuration is required. Configure all NFS clients with fully-qualified domain names to ensure that NFS servers can find clients to notify them of server reboots.

NLM supports advisory file locks only. To lock NFS files, use fcntl(2) with the F_GETLK and F_SETLK commands. The NFS client converts file locks obtained via flock(2) to advisory locks.

When mounting servers that do not support the NLM protocol, or when mounting an NFS server through a firewall that blocks the NLM service port, specify the **nolock** mount option. NLM locking must be disabled with the **nolock** option when using NFS to mount */var* because */var* contains files used by the NLM implementation on Linux.

Specifying the **nolock** option may also be advised to improve the performance of a proprietary application which runs on a single client and uses file locks extensively.

NFS version 4 caching features

The data and metadata caching behavior of NFS version 4 clients is similar to that of earlier versions. However, NFS version 4 adds two features that improve cache behavior: *change attributes* and *file delegation*.

The *change attribute* is a new part of NFS file and directory metadata which tracks data changes. It replaces the use of a file's modification and change time stamps as a way for clients to validate the content of their caches. Change attributes are independent of the time stamp resolution on either the server or client, however.

A *file delegation* is a contract between an NFS version 4 client and server that allows the client to treat a file temporarily as if no other client is accessing it. The server promises to notify the client (via a callback request) if another client attempts to access that file. Once a file has been delegated to a client, the client can cache that file's data and metadata aggressively without contacting the server.

File delegations come in two flavors: *read* and *write*. A*r ead* delegation means that the server notifies the client about any other clients that want to write to the file. A *write* delegation means that the client gets notified about either read or write accessors.

Servers grant file delegations when a file is opened, and can recall delegations at any time when another client wants access to the file that conflicts with any delegations already granted. Delegations on directories are not supported.

In order to support delegation callback, the server checks the network return path to the client during the client's initial contact with the server. If contact with the client cannot be established, the server simply does not grant any delegations to that client.

SECURITY CONSIDERATIONS

NFS servers control access to file data, but they depend on their RPC implementation to provide authentication of NFS requests. Traditional NFS access control mimics the standard mode bit access control provided in local file systems. Traditional RPC authentication uses a number to represent each user (usually the user's own uid), a number to represent the user's group (the user's gid), and a set of up to 16 auxiliary group numbers to represent other groups of which the user may be a member.

Typically, file data and user ID values appear unencrypted (i.e. in the clear) on the network. Moreover, NFS versions 2 and 3 use separate sideband protocols for mounting, locking and unlocking files, and reporting system status of clients and servers. These auxiliary protocols use no authentication.

In addition to combining these sideband protocols with the main NFS protocol, NFS version 4 introduces more advanced forms of access control, authentication, and in-transit data protection. The NFS version 4 specification mandates support for strong authentication and security flavors that provide per-RPC integrity checking and encryption. Because NFS version 4 combines the function of the sideband protocols into the main NFS protocol, the new security features apply to all NFS version 4 operations including mounting, file locking, and so on. RPCGSS authentication can also be used with NFS versions 2 and 3, but it does not protect their sideband protocols.

The **sec** mount option specifies the security flavor that is in effect on a given NFS mount point. Specifying **sec=krb5** provides cryptographic proof of a user's identity in each RPC request. This provides strong verification of the identity of users accessing data on the server. Note that additional configuration besides adding this mount option is required in order to enable Kerberos security. Refer to the rpc.gssd(8) man page for details.

Two additional flavors of Kerberos security are supported: **krb5i** and **krb5p**. The**krb5i** securit y flavor provides a cryptographically strong guarantee that the data in each RPC request has not been tampered with. The **krb5p** security flavor encrypts every RPC request to prevent data exposure during network transit; however, expect some performance impact when using integrity checking or encryption. Similar support for other forms of cryptographic security is also available.

The NFS version 4 protocol allows a client to renegotiate the security flavor when the client crosses into a new filesystem on the server. The newly negotiated flavor effects only accesses of the new filesystem.

Such negotiation typically occurs when a client crosses from a server's pseudo-fs into one of the server's exported physical filesystems, which often have more restrictive security settings than the pseudo-fs.

Using non-privileged source ports

NFS clients usually communicate with NFS servers via network sockets. Each end of a socket is assigned a port value, which is simply a number between 1 and 65535 that distinguishes socket endpoints at the same IP address. A socket is uniquely defined by a tuple that includes the transport protocol (TCP or UDP) and the port values and IP addresses of both endpoints.

The NFS client can choose any source port value for its sockets, but usually chooses a *privileged* port. A privileged port is a port value less than 1024. Only a process with root privileges may create a socket with a privileged source port.

The exact range of privileged source ports that can be chosen is set by a pair of sysctls to avoid choosing a well-known port, such as the port used by ssh. This means the number of source ports available for the NFS client, and therefore the number of socket connections that can be used at the same time, is practically limited to only a few hundred.

As described above, the traditional default NFS authentication scheme, known as AUTH_SYS, relies on sending local UID and GID numbers to identify users making NFS requests. An NFS server assumes that if a connection comes from a privileged port, the UID and GID numbers in the NFS requests on this connection have been verified by the client's kernel or some other local authority. This is an easy system to spoof, but on a trusted physical network between trusted hosts, it is entirely adequate.

Roughly speaking, one socket is used for each NFS mount point. If a client could use non-privileged source ports as well, the number of sockets allowed, and thus the maximum number of concurrent mount points, would be much larger.

Using non-privileged source ports may compromise server security somewhat, since any user on AUTH_SYS mount points can now pretend to be any other when making NFS requests. Thus NFS servers do not support this by default. They explicitly allow it usually via an export option.

To retain good security while allowing as many mount points as possible, it is best to allow non-privileged client connections only if the server and client both require strong authentication, such as Kerberos.

Mounting through a firewall

A firewall may reside between an NFS client and server, or the client or server may block some of its own ports via IP filter rules. It is still possible to mount an NFS server through a firewall, though some of the mount(8) command's automatic service endpoint discovery mechanisms may not work; this requires you to provide specific endpoint details via NFS mount options.

NFS servers normally run a portmapper or rpcbind daemon to advertise their service endpoints to clients. Clients use the rpcbind daemon to determine:

What network port each RPC-based service is using

What transport protocols each RPC-based service supports

The rpcbind daemon uses a well-known port number (111) to help clients find a service endpoint. Although NFS often uses a standard port number (2049), auxiliary services such as the NLM service can choose any unused port number at random.

Common firewall configurations block the well-known rpcbind port. In the absense of an rpcbind service, the server administrator fixes the port number of NFS-related services so that the firewall can allow access to specific NFS service ports. Client administrators then specify the port number for the mountd service via the mount(8) command's **mountport** option. It may also be necessary to enforce the use of TCP or UDP if the firewall blocks one of those transports.

NFS Access Control Lists

Solaris allows NFS version 3 clients direct access to POSIX Access Control Lists stored in its local file systems. This proprietary sideband protocol, known as NFSACL, provides richer access control than mode bits. Linux implements this protocol for compatibility with the Solaris NFS implementation. The NFSACL protocol never became a standard part of the NFS version 3

specification, however.

The NFS version 4 specification mandates a new version of Access Control Lists that are semantically richer than POSIX ACLs. NFS version 4 ACLs are not fully compatible with POSIX ACLs; as such, some translation between the two is required in an environment that mixes POSIX ACLs and NFS version 4.

THE REMOUNT OPTION

Generic mount options such as **rw** and **sync** can be modified on NFS mount points using the **remount** option. See mount(8) for more information on generic mount options.

With few exceptions, NFS-specific options are not able to be modified during a remount. The underlying transport or NFS version cannot be changed by a remount, for example.

Performing a remount on an NFS file system mounted with the **noac** option may have unintended consequences. The **noac** option is a combination of the generic option **sync**, and the NFS-specific option **actimeo=0**.

Unmounting after a remount

For mount points that use NFS versions 2 or 3, the NFS umount subcommand depends on knowing the original set of mount options used to perform the MNT operation. These options are stored on disk by the NFS mount subcommand, and can be erased by a remount.

To ensure that the saved mount options are not erased during a remount, specify either the local mount directory, or the server hostname and export pathname, but not both, during a remount. For example,

mount -o remount,ro /mnt

merges the mount option **ro** with the mount options already saved on disk for the NFS server mounted at /mnt.

FILES

/etc/fstab file system table

BUGS

Before 2.4.7, the Linux NFS client did not support NFS over TCP.

Before 2.4.20, the Linux NFS client used a heuristic to determine whether cached file data was still valid rather than using the standard close-to-open cache coherency method described above.

Starting with 2.4.22, the Linux NFS client employs a Van Jacobsen-based RTT estimator to determine retransmit timeout values when using NFS over UDP.

Before 2.6.0, the Linux NFS client did not support NFS version 4.

Before 2.6.8, the Linux NFS client used only synchronous reads and writes when the **rsize** and **wsize** settings were smaller than the system's page size.

The Linux NFS client does not yet support certain optional features of the NFS version 4 protocol, such as security negotiation, server referrals, and named attributes.

SEE ALSO

fstab(5), mount(8), umount(8), **mount.nfs(5)**, **umount.nfs(5)**, exports(5), netconfig(5), ipv6(7), nfsd(8), sm-notify(8), rpc.statd(8), rpc.idmapd(8), rpc.gssd(8), rpc.svcgssd(8), **kerberos(1)**

RFC 768 for the UDP specification.
RFC 793 for the TCP specification.
RFC 1094 for the NFS version 2 specification.
RFC 1813 for the NFS version 3 specification.
RFC 1832 for the XDR specification.
RFC 1833 for the RPC bind specification.
RFC 2203 for the RPCSEC GSS API protocol specification.
RFC 3530 for the NFS version 4 specification.